JavaScript
Pour débutants

Cours, exemples pratiques et projets pour s'entrainer

Alex Coder

Copyright © Droits d'auteur 2024

Aucune reproduction, stockage dans un système de récupération ou transmission, sous quelque forme que ce soit, ne peut être effectuée sans l'autorisation écrite préalable de l'éditeur, sauf pour de brèves citations intégrées dans des articles critiques ou des critiques.

Tous les efforts ont été déployés pour garantir l'exactitude des informations présentées dans ce livre. Cependant, les exemples contenus dans cet ouvrage sont fournies fonctionnels à l’heure de la rédaction de cet ouvrage dans le contexte de version précise de navigateur et de versions de JavaScript actuelles.

Auteur : Alex Coder
Éditeur : Alex Coder

Première publication : 2024

Remerciements

Cher lecteur, je tiens à vous exprimer ma profonde gratitude pour avoir choisi mon livre. Votre soutien et votre confiance sont une source d'inspiration inestimable.

Merci infiniment d'avoir fait l'acquisition de mon livre. Votre intérêt pour le sujet me motive à continuer à fournir du contenu de qualité.

Je vous invite à télécharger le code source des projets présentés dans ce livre. **Vous trouverez le lien de téléchargement dans la page « Liens de téléchargement » à la fin de ce livre.**

À propos de l’auteur

Alex Coder est un enseignant universitaire, développeur (PHP, JavaScript, Python, Java, C++ etc.) et Data scientist expérimenté dans le domaine des technologies numériques. Fort d'une expertise reconnue, il a dirigé de multiples formations et séminaires internationaux au sein d'institutions de renommée mondiale. Son approche pédagogique exceptionnelle lui a valu une réputation d'excellence parmi ses étudiants, grâce à sa capacité à simplifier des concepts complexes et à les rendre accessibles à tous.

Enseignant universitaire depuis plus de 20 ans, Alex Coder partage sa passion pour l'apprentissage et la transmission des connaissances en informatique, Data science et le Machine Learning. Sa longue expérience dans le domaine éducatif lui permet d'inspirer plusieurs générations d'apprenants à travers le monde, en leur transmettant non seulement des compétences techniques, mais aussi une vision globale du secteur du développement et d’analyse de données.

Passionné par le partage des connaissances, Alex Coder excelle dans la transmission de compétences en informatique, inspirant ainsi plusieurs générations d'apprenants à travers le monde. En plus de ses activités pédagogiques, il offre des formations spécialisées aux professionnels, les aidant à perfectionner leurs compétences et à rester à la pointe de leur domaine respectif.

Son dévouement à l'excellence, sa passion pour l'enseignement et son expertise technique font de Alex Coder une référence incontournable dans le domaine du développement web et de la formation numérique.

Table des matières

Remerciements

À propos de l'auteur

Préface

Introduction et environnement de développement

Chapitre 1: *Votre premier programme JavaScript*

Chapitre 2: *Les variables, constantes et types de données*

Chapitre 3: *Les opérateurs*:

Chapitre 4: *Les structures conditionnelles*

Chapitre 5: *Les boucles en JavaScript*

Chapitre 6: *Les fonctions en JavaScript*

Chapitre 7: *Les tableaux en JavaScript*

Chapitre 8: *Les objets en JavaScript*

Chapitre 9: *Classes et objets*

Chapitre 10: *Principe d'encapsulation en JavaScript*

Chapitre 11: *L'héritage en JavaScript*

Chapitre 12: *Les Prototypes en JavaScript*

Chapitre 13: *Le DOM (Document Object Model)*

Chapitre 14: *Sélection d'éléments dans le DOM*

Chapitre 15: *Modification, ajout et suppression d'éléments du DOM*

Chapitre 16: *Événements dans le DOM*

Chapitre 17: *Projet de développement d'une calculatrice*

Chapitre 18: *La programmation asynchrone en JavaScript*

Chapitre 19: *Requêtes HTTP avec AJAX et Fetch API*

Chapitre 20: *Les API Web*

Chapitre 21: *Projet d'application météo géolocalisée*

Chapitre 22: *Stockage de session en JavaScript*

Conclusion générale

À propos de l’auteur
Lien de téléchargement

Préface

Bienvenue dans l'univers de JavaScript ! Dans ce livre, j'ai souhaité créer un parcours d'apprentissage complet et progressif qui accompagne le lecteur, pas à pas, vers la maîtrise de ce langage essentiel au développement web. JavaScript est bien plus qu'un simple langage de script pour navigateurs. Aujourd'hui, il est partout : sur le front-end, le back-end, et même au-delà des navigateurs. Ce livre a pour ambition de vous offrir une compréhension approfondie de JavaScript, des bases aux concepts avancés.

Nous commençons par une introduction et un aperçu de l'environnement de développement, pour installer les bases nécessaires. Puis, nous plongeons dans les premiers concepts avec le Chapitre 1 où vous découvrirez votre premier programme JavaScript. Ce chapitre est une première prise de contact, un encouragement à plonger dans le code sans crainte.

Ensuite, dans les chapitres suivants, nous explorons les fondamentaux : les variables, constantes, types de données, et les opérateurs. Ces notions sont essentielles pour bâtir la structure de vos programmes, gérer les informations et manipuler les valeurs.

Nous poursuivons avec les structures conditionnelles et les boucles, qui ajoutent de la logique et du dynamisme à vos programmes, vous permettant d'exécuter du code de manière conditionnelle ou répétitive. À travers ces chapitres, vous commencerez à écrire des scripts plus complexes et plus pertinents.

Les fonctions et les tableaux introduits par la suite enrichissent encore vos connaissances, vous offrant des outils puissants pour organiser et manipuler les données. Ces notions constituent le socle de tout programme JavaScript.

Dans la partie suivante, nous abordons la Programmation Orientée Objet (POO). Avec des notions comme les classes, les objets, l'encapsulation, et l'héritage, vous verrez comment structurer votre code de façon modulaire et réutilisable. L'approche orientée objet est souvent redoutée par les débutants, mais vous découvrirez ici une méthode pédagogique pour en faciliter l'assimilation.

Une partie conséquente est dédiée au DOM (Document Object Model).

Interagir avec le DOM vous permet de modifier l'interface de vos pages web en temps réel, d'ajouter ou de supprimer des éléments, et de gérer les événements pour enrichir l'expérience utilisateur.

JavaScript ne se limite pas à la manipulation du DOM ; c'est également un outil puissant pour la programmation asynchrone. Dans les chapitres consacrés à cette thématique, nous abordons des concepts tels que les promesses et l'async/await. Ce sont des notions cruciales pour développer des applications modernes, en particulier celles qui nécessitent des échanges de données en temps réel avec des serveurs.

Les requêtes HTTP, abordées avec AJAX et la Fetch API, vous permettront d'intégrer des données externes dans vos applications. Ce livre ne serait pas complet sans parler des API Web et de la manière de consommer des données depuis des services externes.

Enfin, pour vous aider à mettre en pratique tout ce que vous aurez appris, j'ai inclus deux projets complets : une calculatrice et une application météo géolocalisée. Ces projets sont conçus pour vous offrir une expérience de développement concrète, où vous appliquerez les compétences acquises et gagnerez en confiance.

JavaScript est un langage vivant, dynamique, et en constante évolution. Ce livre est une invitation à explorer ses richesses, à expérimenter, et à développer des projets qui vous passionnent. Que vous soyez débutant ou en phase de consolidation de vos connaissances, j'espère que cet ouvrage sera pour vous une ressource précieuse et inspirante.

Bonne lecture et bon code !

Vous pouvez compter sur mon soutien !

Si vous avez des questions sur un aspect quelconque de ce livre ou si vous avez besoin d'aide, envoyez-moi un e-mail à ***contact@skilldeve.com*** et mentionnez le titre du livre dans l'objet de votre message.

Partagez votre avis

Je suis impatient de connaître votre opinion sur ce livre et je vous en remercie beaucoup.

Introduction et environnement de développement

Le JavaScript est un langage de script côté client dynamique qui permet d'ajouter de l'interactivité et des fonctionnalités avancées aux pages web. À l'origine développé par Netscape Communications Corporation en 1995, il est aujourd'hui omniprésent sur le Web et constitue l'un des piliers du développement front-end.

Exécuté dans le navigateur de l'utilisateur, JavaScript permet de créer une expérience utilisateur engageante et fluide. Avec ce langage, vous pouvez ajouter des effets visuels, manipuler le DOM pour modifier le contenu d'une page de manière dynamique, gérer des formulaires interactifs, afficher des notifications ou encore concevoir des jeux et animations.

Au fil des années, JavaScript a évolué pour devenir plus puissant et polyvalent, notamment grâce à l'introduction d'ECMAScript 6 (ES6), la norme actuelle. ES6 a apporté de nombreuses nouveautés qui simplifient le développement, comme les classes, les modules, les flèches de fonction et les modèles de chaînes de caractères, facilitant ainsi l'écriture de code plus lisible et maintenable.

L'apprentissage du JavaScript se fait progressivement, en commençant par la maîtrise des concepts fondamentaux : variables, types de données, opérateurs, conditions et boucles. Une fois ces bases acquises, vous serez capable de créer des scripts complexes et de les intégrer efficacement dans vos projets web.

Dans les prochains chapitres, nous explorerons en détail ces notions de base, avec des exemples concrets et des exercices qui vous permettront de mieux comprendre comment le JavaScript peut transformer une simple page web en une application dynamique et interactive.

Les Possibilités de JavaScript

Lorsqu'il est utilisé du côté client et exécuté par le navigateur, JavaScript offre une vaste gamme de tâches qui enrichissent l'expérience de l'utilisateur :

Aide à la saisie des formulaires

- Validation des entrées en douceur
- Affichage de messages contextuels clairs
- Édition de texte améliorée

Stockage local sécurisé

- Utilisation des Cookies
- Sauvegarde en local grâce à des technologies telles que localStorage

Gestion des nombres, dates et heures

- Calculs mathématiques puissants
- Affichage des dates adapté à chaque utilisateur
- Animations et graphiques dynamiques
- Menus interactifs et navigation fluide
- Défilements animés, diaporamas et zooms intuitifs
- Conception des Graphiques interactifs et cartes personnalisables
- Développement de Jeux web

Ces capacités ne sont qu'un aperçu du potentiel de JavaScript. Avec quelques bases solides et de la pratique régulière, vous pourrez exploiter toutes les ressources de ce langage pour créer des expériences web engageantes.

Choix et installation d'un environnement de développement intégré

Un environnement de développement intégré, souvent abrégé IDE, est bien plus qu'un simple éditeur de code. Un IDE offre en effet tout un ensemble d'outils pour faciliter le développement d'un projet du début à la fin.

Grâce à l'IDE, un projet complet avec de nombreux fichiers et dossiers peut être chargé, permettant la navigation entre ces éléments. De plus, l'auto-complétion propose des éléments non seulement à partir du fichier courant, mais en tenant compte de l'ensemble du projet. Souvent interconnecté à des systèmes de version tels que Git, l'IDE permet aussi la configuration d'environnements de test et d'autres fonctionnalités essentielles.

Les IDE recommandés

Si vous n'avez pas encore fait votre choix, voici quelques IDE que vous

pouvez envisager pour le développement en JavaScript :

Visual Studio Code[1] : Multiplateforme, gratuit, et extrêmement populaire. Il offre une vaste gamme d'extensions pour enrichir les fonctionnalités de base, notamment pour JavaScript et ses frameworks.

WebStorm[2] : Un IDE de la société JetBrains, spécialement conçu pour le développement frontend. Bien qu'il soit payant, il est très complet et supporte de nombreux langages supplémentaires.

Visual Studio Community[3] (à ne pas confondre avec Visual Studio Code) : Un IDE puissant et payant, spécialement conçu pour les développeurs travaillant sous Windows. Principalement utilisé pour les projets .NET, il n'en demeure pas moins performant pour JavaScript. Une version gratuite, Visual Studio Community, est également disponible pour des projets de plus petite envergure.

Un investissement judicieux

Bien que la plupart des logiciels de développement nécessitent un abonnement payant, de nombreuses options proposent une période d'essai gratuite. Le coût d'une licence est souvent négligeable comparé aux gains de productivité permis, en particulier pour un développeur expérimenté. Cependant, il convient de noter que certains éditeurs totalement libres, comme Visual Studio Code, conviennent parfaitement à la majorité des projets web, y compris ceux en JavaScript. Ces solutions proposent des fonctionnalités avancées, grâce à de nombreux plugins qui étendent leurs capacités sans frais. Dans cet ouvrage, tous les exemples sont créés avec Visual Studio Code, un éditeur gratuit et polyvalent adapté aux besoins de développeurs de tous niveaux. Prenez le temps de choisir l'outil correspondant le mieux à vos besoins et à votre méthode de travail.

Les éditeurs de texte légers

Bien qu'un IDE puisse être très complet, il n'est pas toujours nécessaire de recourir à un tel outil pour chaque tâche mineure. Pour de simples modifications ou l'ouverture rapide d'un fichier, un éditeur de texte léger convient souvent amplement.

Contrairement à un IDE qui fonctionne à l'échelle d'un projet entier (chargement de nombreux fichiers, analyse de la structure, etc.), un éditeur léger se concentre principalement sur la modification d'un ou deux fichiers à la fois, ce qui le rend plus réactif au démarrage et plus

facile à utiliser lorsqu'un environnement complet n'est pas exigé.

Cependant, il est important de noter que les éditeurs légers peuvent aussi être améliorés par le biais de divers plugins. Ces extensions permettent d'ajouter des fonctionnalités telles que l'auto-complétion, la gestion de la syntaxe et bien plus, comblant ainsi en partie l'écart avec un EDI traditionnel.

Quelques éditeurs légers à explorer :

Atom[4] : Multiplateforme et gratuit, avec une grande communauté qui propose de nombreux plugins.
Sublime Text [5]: Également multiplateforme, mais payant. Il est très apprécié pour sa rapidité et son élégance.
Notepad++ [6]: Disponible uniquement sur Windows et gratuit, c'est un choix populaire pour l'édition rapide de fichiers.
Vim[7] **et Emacs**[8] : Des éditeurs très puissants mais demandant un certain apprentissage. Ils sont souvent utilisés par les développeurs expérimentés à la recherche d'un contrôle total sur leur environnement.

Ne discutons pas trop !

Bien que les éditeurs mentionnés précédemment soient excellents, il existe de nombreuses autres options qui méritent d'être considérées. Le choix dépendra de ce que vous cherchez à accomplir, de vos préférences personnelles et des types de projets sur lesquels vous travaillez habituellement. Quelle que soit votre sélection finale, l'essentiel est de vous équiper d'un outil qui augmente votre productivité et s'harmonise naturellement avec votre façon d'écrire du code. Pour d'autres développeurs, un éditeur plus léger ou open source pourrait convenir davantage. Quoi qu'il en soit, l'important est de vous sentir à l'aise pour vous concentrer sur ce qui compte vraiment.

La console de développement

Le code contient inévitablement des erreurs. Et oui, vous allez faire des erreurs, c'est inéluctable, car après tout, vous êtes humain et non une machine ! Mais en tant que développeur, ne vous inquiétez pas, c'est une composante essentielle du processus d'apprentissage.

Les erreurs surviennent malheureusement, mais les navigateurs ne les affichent habituellement pas par défaut. Si une erreur advient, nous ne la verrons pas directement et donc ne saurons où elle se trouve ou comment la corriger. Heureusement, les navigateurs modernes intègrent des outils puissants pour résoudre ce problème : les outils de

développement.

Ces outils sont particulièrement utilisés avec Chrome et Firefox, car ils offrent les fonctionnalités les plus robustes et complètes pour le développement. Bien que d'autres navigateurs possèdent aussi leurs propres outils de développement (souvent avec des particularités intéressantes), Chrome et Firefox demeurent les choix privilégiés des développeurs en raison de leur convivialité et de leur efficacité. Les développeurs choisissent souvent un navigateur “favori” pour le développement, tout en basculant occasionnellement vers d'autres navigateurs pour des tests spécifiques.

Les outils de développement sont extrêmement puissants et offrent une multitude de fonctionnalités. Pour débuter, nous allons apprendre à les ouvrir, à identifier les erreurs dans vos scripts, et à exécuter des commandes JavaScript directement depuis ces outils.

Utilisation des outils de développement avec Google Chrome

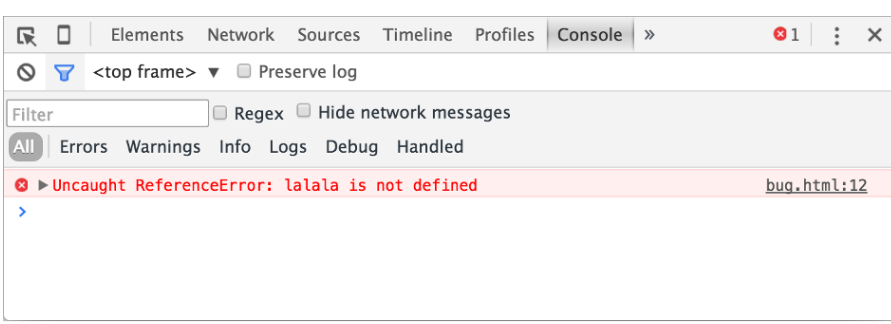
1. Rendez-vous sur la page suivante : bug.html9.

Cette page contient volontairement une erreur dans le code JavaScript, qui ne sera pas visible pour un visiteur classique. Pour révéler cette erreur, nous allons ouvrir la console de développement.

1. Pour ouvrir les outils de développement :
   - Sur Windows : Appuyez sur Ctrl + Maj + J
   - Sur Mac : Utilisez la combinaison Cmd + Option + C

1. Une fois la console ouverte, allez dans l’onglet Console.

Cela vous permettra de visualiser l’erreur et de commencer à la diagnostiquer, comme illustré dans l’image suivante :



Bien que les outils de développement puissent évoluer au fil du temps,

leurs fonctionnalités de base demeurent. Selon la version de google chrome que vous utilisez, l'apparence des outils de développement peut varier.
D'une part, nous repérons un message d'alerte souligné en rouge, révélant un souci dans le code. Ici, la commande "lalala" appelée n'est déclarée nulle part.
D'autre part, en suivant le lien associé, il est possible d'être redirigé directement à la source du problème, en l'occurrence la ligne 12 du fichier bug.html. Ainsi, le problème peut être ciblé avec célérité.
Ces outils, aussi basiques soient-ils, s'avèrent précieux pour débusquer et corriger rapidement les bugs. Qu'ils évoluent en fonction des versions ou non, leur utilité première reste de faciliter la phase de débogage.

bug.html

```
1   <!DOCTYPE HTML>
2   <html>
3
4   <head>
5     <meta charset="utf-8">
6   </head>
7
8   <body>
9
10    There is an error in the script on this page.
11    <script>
12      lalala
13    </script>
14
15  </body>
16
17  </html>
```

Dans la fenêtre de la console Chrome, juste en dessous du message d'erreur, vous pouvez voir un symbole bleu (>)pointant vers le bas. Ce signe indique l'endroit pour entrer des commandes JavaScript directement. Il vous suffit d'appuyer sur la touche Entrée de votre clavier pour exécuter ces commandes immédiatement.
Nous pouvons maintenant visualiser les erreurs, ce qui constitue une première approche convenable. Nous approfondirons l'utilisation des outils de développement et explorerons le débogage en détails dans le chapitre suivant consacré au débogage dans le navigateur.

Saisie multi-lignes

Habituellement, lorsque vous saisissez une ligne de code dans la console et appuyez sur Entrée, cette ligne est exécutée instantanément. Cependant, si vous souhaitez écrire plusieurs lignes avant l'exécution, vous pouvez utiliser le raccourci **Shift** + **Entrée** pour ajouter des

lignes sans déclencher immédiatement le code. Cela permet d'insérer de grands blocs de code JavaScript et de ne les lancer qu'une seule fois une fois terminé.

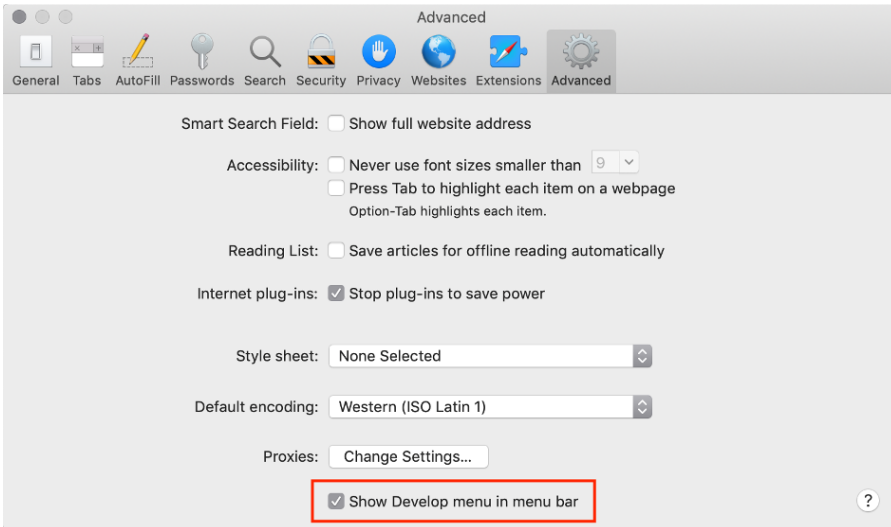
Firefox, Edge et autres navigateurs

La majorité des navigateurs, comme **Mozilla Firefox** et **Microsoft Edge**, permettent d'ouvrir les outils de développement en utilisant la touche **F12**. Bien que globalement similaires dans leur fonctionnement et leur apparence, les subtilités de chaque navigateur signifient qu'il est préférable de maîtriser ces outils individuellement. Je vous suggère de commencer votre apprentissage avec **Google Chrome**, très répandu parmi les développeurs grâce à des outils à la fois perfectionnés et intuitifs. Ce navigateur vous guidera efficacement dans la compréhension des rouages du code. En maîtrisant d'abord Chrome, vous pourrez ensuite facilement explorer les autres alternatives et adapter rapidement vos réflexes de débogage à **Firefox**, **Edge** et consort.

Safari

Safari, le navigateur exclusivement disponible sur Mac, comporte quelques spécificités remarquables. Avant de pouvoir accéder aux outils de développement web, il vous faudra d'abord activer le menu **Développement** au sein des paramètres.

Pour ce faire, ouvrez les Préférences de Safari puis rendez-vous dans l'onglet "**Avancé**" situé tout en bas. Là, cochez simplement la case permettant d'afficher le **menu Développement** directement dans la barre de menus supérieure. Désormais, vous pourrez profiter pleinement des fonctionnalités dédiées aux développeurs web même sur la version desktop du navigateur.



Désormais, plusieurs raccourcis clavier permettent d'activer facilement les outils de développement intégrés aux navigateurs. La combinaison **Cmd** + **Opt** + **C** ouvre directement la console de commande sur **Safari**, ouvrant de nombreuses possibilités pour optimiser et déboguer efficacement les sites et applications Web.
Les développeurs disposent également d'un menu dédié regroupant l'ensemble des fonctionnalités avancées : inspection des éléments, exécution de code JavaScript, surveillance des performances et bien d'autres sont accessibles en un clic. Quel que soit l'environnement ou le navigateur, les outils de développement offrent une visibilité totale permettant de corriger les bugs et améliorer l'ergonomie des interfaces.
En résumé :

- Sur **Windows**, appuyer sur **F12** ouvrira généralement l'inspecteur dans la plupart des navigateurs.
- Sous **MacOS**, la touche **Cmd** + **Opt** + **J** permet d'activer les outils de développement dans Chrome de manière immédiate.
- Pour **Safari**, il suffit de saisir **Cmd** + **Opt** + **C** après avoir activé le menu Développement dans les paramètres.

Conclusion

Au cours de ce premier chapitre introductif, nous avons exploré les bases de JavaScript, un langage de programmation essentiel pour ajouter de l'interactivité aux pages web. Nous avons brièvement discuté de son histoire, de sa popularité et de ses principales utilisations dans le développement web moderne.

Ensuite, nous avons découvert l'importance de l'environnement de développement. Nous avons expliqué ce qu'est un IDE (environnement de développement intégré) et présenté les différents outils que vous pouvez utiliser pour écrire et déboguer votre code JavaScript, notamment Visual Studio Code, que nous utiliserons tout au long de ce livre.

Les outils de développement des navigateurs, tels que ceux disponibles dans Google Chrome, Firefox et Safari, ont également été introduits. Ils vous permettent de visualiser les erreurs, de tester votre code, d'inspecter des variables et bien plus encore.

Enfin, nous vous avons montré comment configurer ces outils dans les navigateurs et comment les utiliser efficacement pour déboguer des scripts JavaScript. Vous disposez désormais de tout ce dont vous avez besoin pour commencer à coder.

Dans les prochains chapitres, nous allons aborder le cœur du sujet en examinant les concepts fondamentaux de JavaScript et en mettant en pratique ce que nous avons appris.

Chapitre 1

Votre premier programme JavaScript

Bienvenue pour votre premier pas avec JavaScript! Ce chapitre amorce votre exploration de ce langage primordial pour le développement web interactif. JavaScript anime vos pages internet, leur conférant dynamisme et interactivité.

Au cours de ce chapitre, nous vous guiderons étape par étape pour écrire votre tout premier programme JavaScript. Vous apprendrez à intégrer du code JavaScript dans une page HTML, que ce soit directement dans le document html ou via des fichiers externes. Nous aborderons également quelques bonnes pratiques pour organiser votre code de manière efficace.

À l'issue de ce chapitre, vous saurez créer et exécuter de simples scripts JavaScript et comprendrez comment les intégrer à un projet web. Préparez-vous à écrire vos premières lignes de code et à découvrir la puissance de JavaScript

Intégrer JavaScript à une page web

Tout comme les feuilles de style CSS, il existe deux approches principales pour intégrer du JavaScript à vos pages web. La première consiste à directement inscrire le script au sein même du document, tandis que la seconde repose sur le rattachement d'un fichier externe au moyen de liaisons. Dans les deux cas, l'élément script sert de support.

Méthode 1 : Script intégré dans le document

Premièrement, il est possible d'insérer directement le code entre les balises script, à l'intérieur même de la structure HTML, PHP ou autre. Ainsi en va-t-il de ce script élémentaire :

```
<script>
          // Votre code JavaScript ici
          console.log('Bonjour, monde !');
 </script>
```

Cette solution ressort pratique pour de courts scripts ou des besoins ponctuels.

Méthode 2 : Scripts externes

Pour des projets plus élaborés cependant, il est recommandé d'externaliser le JavaScript au sein de documents dédiés, auxquels on se rattache au moyen de l'attribut **src**. Prenons l'exemple d'un fichier my_script.js. Voici un exemple pour pointer vers ce fichier JavaScript à partir d'une page HTML.

```
  <script src="my_script.js"></script>
```

Bien que les scripts externes offrent l'avantage de réutiliser du code sur plusieurs pages du site, tout comme les feuilles de style externes, leur utilisation n'est pas sans défaut. En effet, chaque fichier externe appelé nécessite une requête supplémentaire auprès du serveur, alourdissant d'autant le chargement de la page pour l'utilisateur. Malgré cette contrainte liée à la vitesse de chargement, inclure le code JavaScript dans des fichiers séparés présente de nombreux bénéfices. Outre la réduction de la redondance, cette méthode de programmation favorise aussi la maintenance et l'évolutivité du site.

Où placer les scripts dans votre document html?

Bien que les développeurs puissent insérer le script n'importe où sur une page web, deux endroits sont privilégiés :

Dans l'en-tête, avant la fermeture de la balise **<head>.** De cette façon, le code javascript se chargera avant l'affichage complet de la page, mais il risque de ralentir le rendu initial.

Juste avant la fermeture de la balise **<body>,** tout en bas de la structure HTML. Cet emplacement est le plus adapté car il permet au navigateur de parser et d'analyser entièrement le contenu avant d'exécuter le script. Ainsi, la satisfaction de l'utilisateur lors du chargement initial s'en trouve améliorée.

Exemple de script externe placé en bas de page :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Ma première page JavaScript</title>
</head>
<body>
    <!-- Contenu de la page -->

    <script src="my_script.js"></script>
</body>
</html>
```

Pourquoi placer les scripts en bas de la page ?
Bien que positionner les scripts au bas de la page offre certains avantages, ce n'est pas toujours la solution idéale. Chargeant le JavaScript en dernier permet au navigateur de construire la structure DOM (Document Object Model) avant l'exécution du code, ce qui peut accélérer les performances dans certains cas. Cependant, des éléments nécessitant du JavaScript risquent de ne pas s'afficher tant que le script n'est pas téléversé. De plus, certains scripts peuvent être nécessaires tôt dans le processus de rendu pour optimiser l'expérience utilisateur. Il vaut mieux équilibrer les priorités en fonction du type de page.

Remarques importantes :

- Scripts simples dans le document HTML, scripts complexes externalisés : Il est en général recommandé de conserver les scripts élémentaires dans le document HTML. Pour des scripts plus élaborés, ayez recours à des fichiers externes afin d'améliorer l'organisation.
- Mise en mémoire cache des fichiers externes : Dès qu'un fichier JavaScript externe est téléchargé par le navigateur, il est stocké dans le cache. Si d'autres pages ont recours à ce même script, il sera chargé directement depuis la mémoire cache sans devoir être à nouveau téléchargé. Cela permet d'économiser la bande passante et d'accélérer le chargement des pages.
- Un script à la fois : Lorsque l'attribut src est inclus dans un élément, le contenu de cette balise est ignoré. Vous ne pouvez pas avoir à la fois du code JavaScript à l'intérieur de la balise et l'attribut src en même temps.

Structure du code

La première étape pour apprendre JavaScript nécessite de bien saisir comment structurer adéquatement les blocs de code.

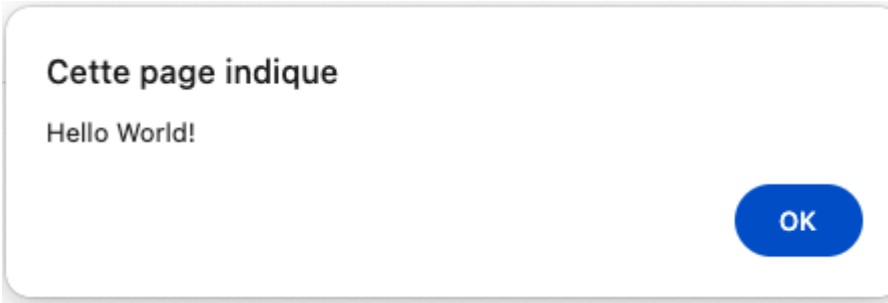
Les instructions

Les instructions en JavaScript sont des ordres qui exécutent des actions précises. Par exemple, l'instruction qui suit affiche un message à l'écran :

```
alert("Hello World!");
```

Cette ligne de code représente une instruction mettant en œuvre une action : afficher une boîte de dialogue contenant le message "Hello World!".

Si vous placez ce code dans un fichier nommé my_script.js et que vous reliez ce document à une page index.html, l'exécution de cette page déclenchera l'affichage d'une fenêtre d'alerte, comme illustré dans l'exemple suivant :



Bien que la syntaxe JavaScript autorise certaines flexibilités, il est préférable d'adopter une structure claire et explicite dans le code. L'ajout systématique de points-virgules à la fin de chaque instruction permet d'éviter les erreurs d'interprétation lors de l'exécution du programme.

Prenons l'exemple d'un script devant afficher deux messages successifs. Nous pourrions être tentés d'écrire :

```
alert("Hello");
alert("World");
```

Ces deux lignes enchaînent deux appels à la fonction alert(). Cependant, sans caractère délimitateur, le moteur JavaScript pourrait considérer ces instructions comme formant un bloc et tenter de les exécuter ensemble. L'ajout d'un point-virgule après chaque ligne lève toute ambiguïté.

Bien que le standard autorise parfois l'omission du point-virgule, il est donc préférable, par souci de clarté et de robustesse, d'en insérer systématiquement à la fin de chaque instruction. Cette bonne pratique évite les bugs logiciels en garantissant une compréhension univoque du code par l'interpréteur.

```
alert("Hello")
alert("World")
```

Bien que JavaScript insère habituellement des points-virgules de manière automatique lorsqu'on revient à la ligne, cela ne fonctionne pas dans toutes les situations. En effet, dans certains cas précis, le moteur d'exécution ne parvient pas à déterminer de façon univoque où placer le point-virgule manquant. Par exemple, lorsque plusieurs instructions s'enchaînent sur la même ligne de code sans séparateur explicite, l'interpréteur se retrouve désemparé. Il importe donc pour les développeurs de ne pas trop se reposer sur ce mécanisme implicite

et de veiller à ce que la syntaxe du programme soit toujours parfaitement définie. Une petite inattention à ce niveau peut vite mener à des bugs difficiles à détecter.

Exemple :

```
alert(3 +
    1 +
    2);
```

Bien que JavaScript puisse paraître simple à première vue, sa syntaxe respecte des règles précises. Lors de l'écriture du code, le moindre détail compte. Prenons cet exemple où une expression n'est que partiellement renseignée : au lieu de lever une erreur, le navigateur compilera la ligne de manière tacite.
JavaScript a compris que les trois lignes constituent la même instruction et le code s'exécute correctement.
Voyons les points clés à retenir concernant la syntaxe :
- JavaScript distingue minuscules et majuscules dans les identifiants. Une attention soutenue aux noms choisis s'impose.
- Les sauts de ligne et espaces ne sont pas significatifs, si ce n'est dans les chaînes littérales. Ils aèrent seulement le code.
- Tout script s'organise en une suite d'instructions qui guident les navigateurs pour exécuter les actions. Même un simple message d'alerte nécessite une déclaration valide.
Au-delà des apparences, la maîtrise de JavaScript requiert une connaissance approfondie de ses fondements syntaxiques. Seul ce socle permet d'écrire un code lisible, maintenable et exempt d'erreurs.

```
  alert("Merci.");
```

Le point-virgule à la fin de cette déclaration informe le langage JavaScript que l'instruction est terminée, fonctionnant de manière similaire à une phrase linguistique s'achevant par un point final. Bien que JavaScript puisse considérer une nouvelle ligne comme une fin d'instruction, il est préférable d'utiliser systématiquement des points-virgules afin d'éviter toute équivoque quant à la terminaison correcte. De plus courtes instructions sur une ligne unique peuvent se combiner à de plus longues phrases s'étirant sur plusieurs lignes, cette alternance permettant de rompre la monotonie sans nuire à la compréhension ou au bon fonctionnement du code.

Les commentaires

Les annotations présentes dans le code JavaScript permettent d'ajouter des remarques qui ne seront pas prises en compte par le navigateur lors de l'exécution du script. Elles sont particulièrement utiles pour documenter le code, éclaircir certains passages ou laisser des notes à l'intention d'autres développeurs.

Il existe deux types d'annotations :

Commentaires sur une seule ligne : Commençant par //, le reste de la ligne est ignoré par JavaScript.

Exemple :

```
// Ceci est un commentaire sur une seule ligne.
```

Commentaires sur plusieurs lignes : Débutant par /* et se terminant par */, tout le texte compris entre ces deux balises sera passé sous silence.

Exemple :

```
/* Ceci est un commentaire sur
plusieurs lignes */
```

Raccourcis pour commenter
Les éditeurs de code offrent souvent des moyens simples d'ajouter des commentaires de manière temporaire ou permanente. Il est par exemple possible de commenter une seule ligne en utilisant des raccourcis clavier comme **Ctrl** + / sous **Windows** ou **Cmd** + / sous **Mac**.
Pour des blocs de lignes, la manipulation est similaire mais utilise **Ctrl** + **Shift** + / ou **Cmd** + **Option** + / selon le système. Ces options pratiques permettent de désactiver rapidement des portions de code lors des phases de test ou d'y ajouter des notes sans modifier le code source. Les développeurs apprécient ces fonctionnalités car elles accélèrent la mise au point et facilitent la collaboration. Bien que basiques, les commentaires restent une composante essentielle de tout projet informatique pour documenter et expliquer le fonctionnement d'un programme.

Conclusion

Félicitations ! Après beaucoup de travail acharné, vous avez réussi à exécuter votre tout premier programme JavaScript. Bien que vous ayez encore beaucoup à apprendre, n'oubliez pas à quel point vous avez progressé depuis le début.

Au début, les bases du langage comme l'intégration du script dans une page web ou la gestion des points-virgules pouvaient sembler compliquées. Maintenant, vous maîtrisez ces concepts essentiels et êtes prêt à explorer des sujets plus avancés.

Les prochains chapitres approfondiront ces connaissances de base et vous présenteront de nouveaux défis stimulants. Vous apprendrez à manipuler des variables, créer des fonctions et rendre vos pages web encore plus interactives. N'hésitez pas à vous lancer dans de nouveaux projets pour consolider vos compétences.

Souvenez-vous que la programmation demande de la patience et de la pratique. Si parfois certains concepts vous semblent obscurs, n'abandonnez pas. Prenez le temps de réfléchir et d'expérimenter - avec de la persévérance, vous surmonterez tous les obstacles. Bonne chance dans votre apprentissage!

Chapitre 2

Les variables, constantes et types de données

Au cours de ce chapitre, nous explorerons l'un des fondements essentiels de JavaScript : la gestion des données. Pour qu'un programme puisse effectuer des opérations et manipuler des informations, il est impératif d'apprendre à stocker et organiser ces éléments. C'est là qu'interviennent les variables, les constantes et les types de données.

Les variables permettent d'emmagasiner des valeurs susceptibles d'évoluer durant l'exécution du programme, tandis que les constantes stockent des valeurs fixes. JavaScript propose diverses catégories de données pour représenter des informations variées comme les nombres, les chaînes de caractères, les objets et bien d'autres encore.

Au cours de ce chapitre, vous apprendrez à :

- Déclarer et utiliser des variables ainsi que des constantes.
- Comprendre les principaux types de données fondamentaux en JavaScript.
- Manipuler les données de façon efficace dans vos programmes.

Ces notions constituent la base de tout développement en JavaScript et sont indispensables pour créer des applications web interactives. Êtes-vous prêt à commencer ? Partons à la découverte de ces concepts !

Les Variables en JavaScript

À l'instar d'un conteneur, une variable emmagasine une donnée. Lorsque vous lui attribuez un nom suivi d'une valeur numérique, chaîne de caractères, fonction ou objet complexe, vous pouvez manipuler et réutiliser facilement cette information tout au long de votre script. De plus, la valeur d'une variable demeure mutable, évoluant au gré de la logique de votre programme.

En Javascript, trois façons principales existent pour déclarer des variables : au moyen de var, **let** et **const**. Concentrons-nous sur les deux premiers.

Cas 1 : Utilisation du mot-clé var
Le mot-clé **var** présent depuis les débuts de Javascript en 1995, illustre simplement sa fonctionnalité. Voici un exemple :

```
var number1 = 22;
var number2 = 50;
var number = number2 - number1;
alert("La valeur du number est = " + number);
```

Dans cet exemple :

- La variable number1 a la valeur 22.
- La variable number2 a la valeur 50.
- La variable number est calculée en soustrayant number1 de number2, ce qui donne 28.

Bien que le mot clé "var" reste valide, il est souvent remplacé par des méthodes plus modernes que nous verrons sous peu. Notez qu'il est possible de nommer vos variables comme vous le souhaitez tout en respectant certaines conventions :

L'exécution de ce code aboutit à l'affichage de la boite d'alerte suivante :

Le nom doit débuter par une lettre, un trait de soulignement ou un signe dollar. Il peut contenir des lettres, chiffres, traits de soulignement ou dollars mais ne doit comporter ni espaces ni caractères spéciaux. La casse est importante : myVariable, myvariable et MYVariable sont trois variables distinctes.

Par exemple :

```
var userAge;
var USERage;
var USerAGE;
```

Dans cet exemple, nous avons défini trois variables aux noms similaires pour observer leur portée et valeur. Avec « userAge », « USERage », et « USerAGE », nous verrons que JavaScript fait la distinction entre les majuscules et les minuscules.

Cas 2 : Utilisation du mot-clé let

Introduit dans ES6 en 2015, le mot clés « **let** » remplace progressivement « **var** » pour déclarer de nouvelles variables de manière plus moderne. Corrigeant certains pièges de « var », « let » délimite mieux la portée des variables.

Dans l'exemple ci-dessous, « let » produit le même résultat que « var » pour déclarer une variable.

```
let number1 = 22;
let number2 = 50;
let number = number2 - number1;
alert("La valeur du number est = " + number);
```

L'exécution de ce code aboutit au même résultat obtenu lorsque nous avons utilisé le mot clé « var ».

Bien que cela puisse nuire à la lisibilité, il est aussi tout à fait possible de déclarer plusieurs variables sur une seule ligne pour réduire la longueur de votre code,:

```
let number1 = 22, number2 = 50, number = number2 - number1;
alert("La valeur du number est = " + number);
```

Notez bien qu'il est préférable d'aligner vos déclarations sur plusieurs lignes pour un code plus lisible :

```
    let number1 = 22,
        number2 = 50,
        number = number2 - number1;
    alert("La valeur du number est = " + number);
```

Bien que var et let servent tous deux à déclarer des variables en JavaScript, ils diffèrent notablement en termes de portée. Contrairement à **var** dont la portée est globale ou fonctionnelle, **let** ne définit une variable que dans le bloc où elle est introduite.

Modification des valeurs de variables

La capacité d'une variable à évoluer au fil du programme est essentielle. Quel que soit le nombre de réaffectations dont elle fait l'objet, seule sa valeur la plus récente importe pour la suite de l'exécution. Prenons l'exemple suivant :

```
    let number1 = 10;
    let number2 = 50;
    number1 = 20; // La valeur de number1 est désormais 20
    let number = number1 + number2;
    alert("La valeur du number est = " + number);
```

Si vous exécutez ce programme vous obtiendrez une boite d'alerte qui affiche la phrase : « La valeur du number est = 70 »

Vous pouvez aussi affecter la valeur d'une variable à une autre variable :

```
let number1 = 10;
let number2 = 50;
number2 = number1; // number2 prend la valeur de number1
let number = number1 + number2;
alert("La valeur du number est = " + number);
```

Ici, le résultat sera 20 car number2 prend la valeur de number1 (10), et la somme des deux variables donne 20.

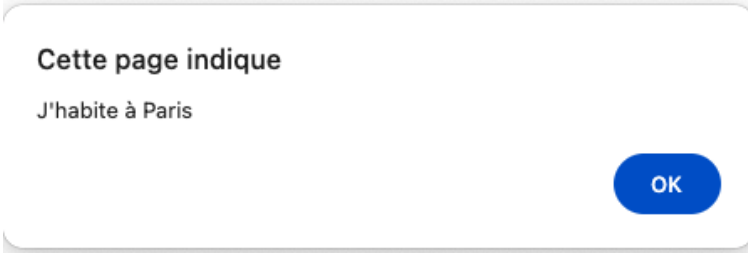
Exemple avec des chaînes de caractères
Les variables peuvent également contenir des chaînes de texte. Voici un exemple simple :

```
let city1 = "Paris";
let city2 = "Lyon";
let city = city1;
alert("J'habite à " + city); // Affiche : J'habite à Paris
```

- Le programme commence par définir deux variables de type chaîne de caractères appelées **city1** et **city2**. **city1** se voit attribuer la valeur "Paris" tandis que city2 prend la valeur "Lyon".
- Une troisième variable nommée **city** est ensuite déclarée. Celle-ci va prendre la valeur stockée dans **city1**, à savoir "Paris". La ville contenue dans city sera donc Paris.
- Pour finir, une alerte JavaScript s'affiche à l'écran. Son message résulte de la concaténation de la chaîne "**J'habite à** " et du contenu de la variable city. Puisque city vaut "Paris", le message complet affiché sera

"J'habite à Paris".

Résultat :

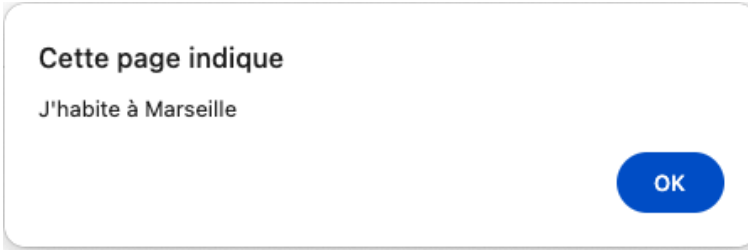


Vous pouvez ensuite réaffecter la valeur d'une variable à partir d'une autre :

```
    let city1 = "Paris";
    let city2 = "Lyon";
    let city3 = "Marseille";
    city1 = city3; // city1 prend la valeur de city3
    let city = city1;
      alert("J'habite à " + city); // Affiche : J'habite à
Marseille
```

Explication :
Tout d'abord, trois variables représentant des villes ont été déclarées de façons distinctes: city1 a reçu la valeur "Paris", city2 "Lyon" et city3 "Marseille".
Par la suite, une modification a été apportée : la valeur initiale de city1, soit "Paris", a été remplacée par la valeur de city3, c'est-à-dire "Marseille". Dès lors, city1 ne contenait plus la capitale française mais bien la ville de Marseille.
Puis, une nouvelle variable nommée city a vu le jour. Celle-ci s'est vue assigner la valeur de city1, à ce moment "Marseille". Les deux variables partageaient désormais la même valeur.
Finalement, un message d'alerte fut déclenché affichant la phrase "J'habite à" suivie de la valeur emmagasinée dans city. Puisque cette dernière était "Marseille", le résultat final fut "J'habite à Marseille", dévoilant ainsi la ville de résidence désignée par le truchement des manipulations opérées sur les variables initiales.
En somme, ce code a commencé par trois variables représentant des villes françaises. city1 a ensuite échangé sa valeur avec celle de city3. city s'est alors vu attribuer la valeur de city1. L'alerte a donc

logiquement affiché "J'habite à Marseille".



Types de données

Les données stockées dans les variables JavaScript peuvent prendre diverses formes. Il est essentiel de distinguer ces types de valeurs pour manipuler les informations de manière appropriée dans nos programmes.
L'opérateur **typeof** nous permet de connaître précisément la nature d'une valeur. Cet outil s'avère très utile pour traiter les données de façon personnalisée ou effectuer des contrôles rapides.

L'opérateur typeof
Cet opérateur renvoie le type de la valeur passée en paramètre, sous la forme d'une chaîne de caractères. Il existe deux syntaxes équivalentes pour l'utiliser :

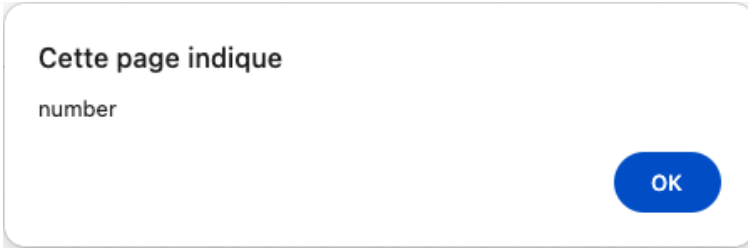
- Forme opérateur : **typeof x**
- Forme fonction : **typeof(x)**

Les nombres
Les variables peuvent contenir des nombres à virgule ou à partie entière. Par exemple :

```
let number = 105;
alert(typeof number); // Affiche "number"
```

En JavaScript, le type de variable est automatiquement déterminé en fonction de la valeur qui lui est attribuée. Ici, un nombre entier est associé à la variable "number", dont le type est précisément number.

Lorsque le code précédent est exécuté, une boîte d'alerte s'affiche et indique que le type de la variable "number" est bien number. De façon surprenante, JavaScript identifie de lui-même le type de données stocké dans chaque variable, ce qui facilite grandement le développement en évitant d'avoir à le spécifier manuellement à chaque fois. Le langage reste néanmoins très souple et tolérant, autorisant les changements de types en fonction des opérations effectuées sur les variables.



Exemple avec un nombre à virgule flottante :

```
let number = 42.8;
alert(typeof(number)); // Affiche "number"
```

Le langage JavaScript déterminera automatiquement le type des variables en fonction des valeurs, vous n'avez pas besoin de préciser les types de données de manière explicite.

Les opérateurs d'addition (+), de soustraction (-), de multiplication (*) et de division (/) sont utilisés pour les calculs numériques de base. Ces opérateurs arithmétiques de base s'appliquent de manière transparente quelle que soit la valeur, le langage déterminant toujours le type approprié en fonction du contexte et des données saisies.

Exemple d'addition de deux nombres :

```
let number = 105;
let somme = number + number;
alert(somme); // Affiche 210
```

Les chaînes de caractères (String)

Une chaîne de caractères est une séquence de lettres, chiffres ou autres caractères entourée de guillemets simples ou doubles. Voici un exemple simple :

```
let city = "Paris";
alert(typeof city); // Affiche "string"
```

Dans cet exemple, nous avons :

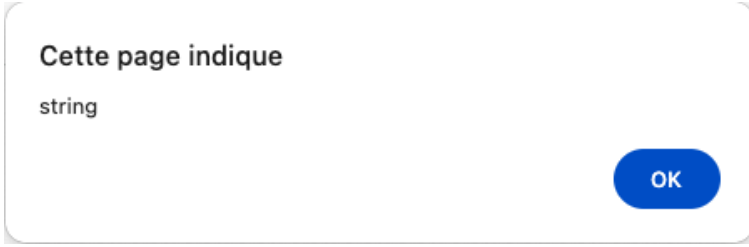
- La valeur "Paris" est une chaîne de caractères (ou string en anglais), car elle est entourée de guillemets.

- La fonction **typeof city** permet de connaître le type de la variable city. Dans ce cas, comme city contient la chaîne de caractères "Paris", le type renvoyé par typeof sera "string".

Vous pouvez également utiliser des chaînes de caractères plus longues :

```
    let expression = "J'ai commencé ma formation il y a 2 jours";
    alert(typeof expression); // Affiche "string"
```

Ce code déclare une variable « expression » contenant une chaîne de caractères (une phrase) et utilise l'opérateur typeof pour confirmer que cette variable est bien de type string. Le résultat "string" est ensuite affiché dans une boîte d'alerte.



Notez que si vous affectez un nombre entre guillemets à une variable, il sera traité comme une chaîne de caractères, et non comme un nombre :

```
    let number = "10";
    alert(typeof(number)); // Affiche "string"
```

Lorsque vous exécutez ce code vous obtiendrez une boite d'alerte qui affiche que le type de la variable « number » est string (chaîne de caractères).

La concaténation

Le signe + peut également être utilisé pour concaténer (coller) des chaînes de caractères ensemble. Par exemple :

```
    let city1 = "Paris";
    let city2 = "Marseille";
```

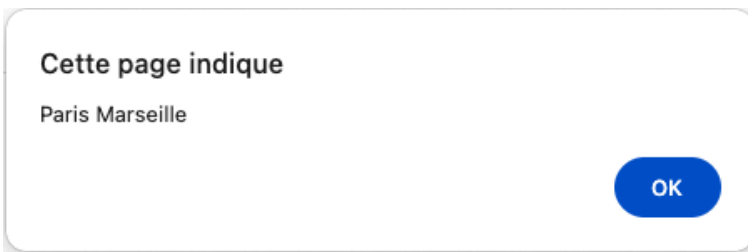
```
alert(city1 + " " + city2); // Affiche "Paris Marseille"
```

Explication :

city1 + " " + city2 : Cette expression permet de concaténer (c'est-à-dire regroupe) plusieurs chaînes de caractères.

- La variable city1 contient "Paris",
- " " est un espace,
- La variable city2 contient "Marseille".

En utilisant l'opérateur +, JavaScript combine ces trois éléments pour former une seule chaîne : "Paris Marseille".

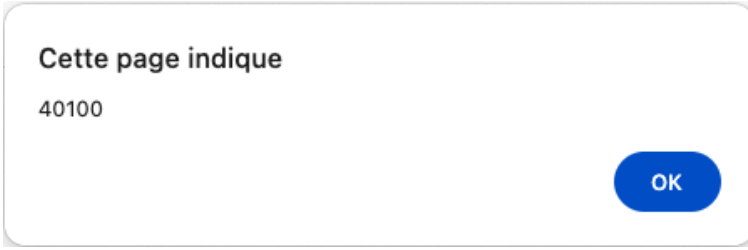


Lorsque vous additionnez deux chaînes, elles sont simplement collées l'une à l'autre. Cela s'applique également aux chaînes qui contiennent des chiffres :

```
let number1 = "40";
let number2 = "100";
alert(number1 + number2); // Affiche "40100"
```

Ce script déclare deux variables contenant des chaînes ressemblant à des valeurs numériques. Lors de l'utilisation de l'opérateur de concaténation, JavaScript relie les chaînes de caractères pour former une nouvelle chaîne "40100", qui est ensuite affichée dans une alerte.

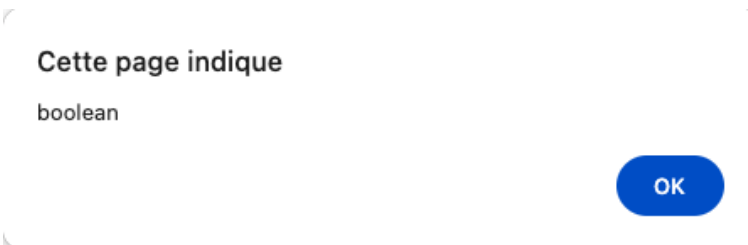
En réalité, au lieu d'additionner 40 et 100 comme des nombres, le langage relie littéralement les deux chaînes "40" et "100", donnant "40100". Ainsi, l'addition ne se fait pas de manière arithmétique mais textuelle, démontrant la subtilité des types de données en JavaScript.



Les booléens (Boolean)
Un booléen est une variable qui ne peut avoir que deux valeurs : true (vrai) ou false (faux). Ces valeurs sont souvent utilisées dans les conditions.

```
let truth1 = true;
let truth2 = false;
alert(typeof truth1); // Affiche "boolean"
```

Ce code déclare deux variables booléennes, truth1 (avec la valeur true) et truth2 (avec la valeur false). Ensuite, il utilise l'opérateur typeof pour vérifier que truth1 est de type booléen, et ce type est affiché dans une alerte.



Si vous placez true ou false entre guillemets, elles seront interprétées comme des chaînes de caractères et non comme des booléens :

```
let truth1 = "true"; // Chaîne de caractères
let truth2 = false; // Booléen
alert(typeof truth1); // Affiche "string"
alert(typeof truth2); // Affiche "boolean"
```

Le type indéfini (undefined)

Une variable déclarée mais à laquelle aucune valeur n'a été assignée est du type undefined :

```
let name;
alert(typeof name); // Affiche "undefined"
```

La variable désignée par "name" a été déclarée, mais ne s'est vu attribuer aucune valeur. Par conséquent, son contenu demeure indéterminé.

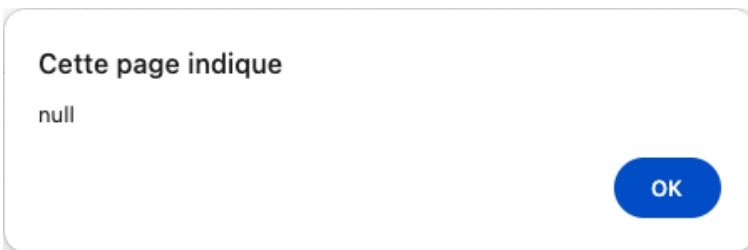
Lorsqu'appliquée à une variable non initialisée ou explicitement définie comme vide, la fonction "typeof" renvoie systématiquement "undefined". C'est pourquoi l'instruction "alert(typeof name);" affiche ce terme dans le cas présent.

Le type nul (null)

Le type nul, noté "null", désigne une valeur spéciale employée pour marquer l'intention délibérée de ne rien spécifier. Contrairement à "undefined" qui souligne l'indéfinition, "null" souligne la vacuité ou la nullité.

```
let name = null;
alert(name); // Affiche null
```

La variable name est initialisée à null, ce qui signifie que sa valeur est explicitement définie comme "aucune valeur" ou "vide".
Lorsque vous affichez « name » avec alert(), vous obtiendrez la valeur null.



Autres types de données
JavaScript offre également des types de données plus élaborés comme les tableaux et les objets, que nous explorerons dans un chapitre à venir.

Constantes en JavaScript

Tel que leur nom l'indique, une constante représente une entité dont la valeur demeure inchangeable une fois définie. En d'autres mots, une constante conserve la même valeur tout au long de l'exécution du programme. Contrairement à une variable, la valeur d'une constante ne peut être modifiée après sa déclaration initiale.

Les constantes sont principalement utilisées pour stocker des valeurs qui doivent demeurer fixes, comme des paramètres de configuration, des clés d'API ou des constantes mathématiques.

En résumé, une constante correspond à une variable dont la valeur est immuable. Elle est définie une seule fois et ne peut être altérée.

Définir et utiliser une constante

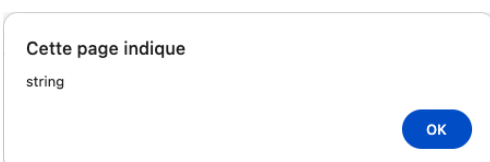
Pour déclarer une constante en JavaScript, on a recours au mot-clé **const**, introduit dans la norme ES6 (2015). Ce mot-clé permet de créer une constante dont la valeur ne pourra être modifiée après son initialisation.

Voici un exemple de déclaration et d'utilisation d'une constante:

```
const name = "Amine";


alert(name);
alert(typeof name);
```

Explication du code :
En exécutant ce code, la première alerte (alert(name)) affiche la valeur de la constante « name », qui est "Amine".
Quant à la deuxième alerte (alert(typeof name)) affiche le type de la variables « name ». Puisque « name » est une chaîne de caractères (string), le type affiché sera "string".

Lorsque vous exécutez ce code vous obtiendrez successivement les boites d'alerte suivantes :



Tentative de modification d'une constante
Essayons maintenant de réaffecter une valeur à une constante après sa déclaration :

```
const name = "Amine";
name = "Jacques"; // Tentative de modification
alert(name);
```

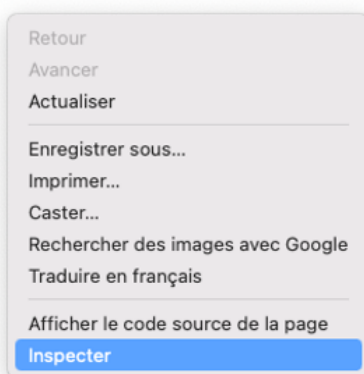
Que se passe-t-il ?

Dans ce cas particulier, un problème apparaîtra. JavaScript détecte la tentative de modification d'une constante et génère une erreur au moment de l'exécution.

Résultat : la page ne s'affiche pas correctement et aucun avertissement n'est affiché.

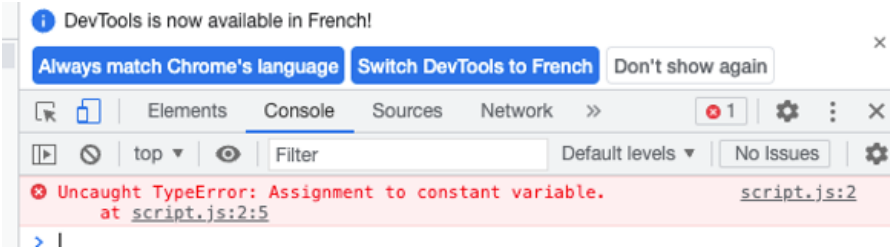
Analyser les erreurs

Si la page est entièrement blanche sans messages d'alerte, il est probable qu'une erreur s'est produite. Dans cette situation, vous pouvez ouvrir la console de développement de votre navigateur (par exemple dans Chrome, en appuyant sur Ctrl + Shift + J ou Cmd + Option + J sur macOS).
Tu peux aussi faire un clic droit -> inspecter -> console



L'erreur que vous trouverez dans la console de votre navigateur sera probablement semblable à :

TypeError: Assignment to constant variable.



Cela signifie que tu as tenté de modifier une constante, ce qui est interdit en JavaScript.

La pédagogie de l'erreur

L'analyse des erreurs est une compétence essentielle pour tout développeur. Bien que les bugs fassent partie intégrante du processus de codage, il importe de savoir les identifier et comprendre leur origine. Chaque erreur représente une opportunité d'apprentissage pour affiner ses aptitudes. Ne te décourage pas des imperfections ; elles témoignent de ton cheminement vers la maîtrise optimale.

En résumé :

- Une constante se voit assigner une valeur fixe qu'il n'est pas loisible de modifier par la suite. Toute tentative de réaffectation génère une exception.
- Le caractère « immuable » d'une constante se limite à sa valeur initiale, non aux objets ou tableaux qui lui sont liés.
- Le recours aux constantes permet d'assurer l'intégrité de données cruciales tout au long de l'exécution.
- Les erreurs de réassignation d'une constante traduisent la rigueur propre à JavaScript, garante d'un comportement prévisible et d'un code fiable.

Var et let : quelles différences ?

En JavaScript, **var** et **let** sont deux moyens de déclarer une variable, mais ils diffèrent en ce qui concerne la portée, leur hoisting et certain comportement dans de tels contextes. Regardons les principales différences entre eux.

Portée (Scope)
La portée de **var** est **fonctionnelle** ou **globale**: cela signifie quand vous déclarez une variable avec **var** dans la même fonction, par exemple, la variable peut être utilisée partout à l'intérieur de cette fonction. Mais, si vous la déclarez en dehors d'une fonction, cette variable sera accessible de manière globale.

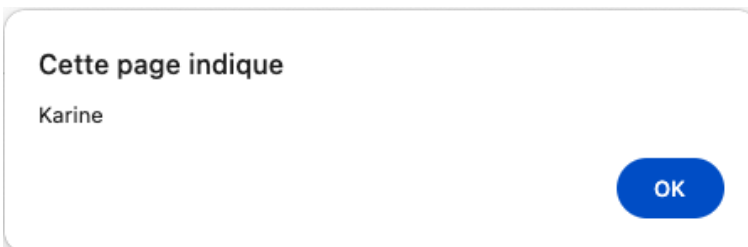
Let est une portée de bloc : une variable déclarée par **let** n'est pas disponible que dans le bloc où elle est définie. Les blocs peuvent être des blocs de if, de boucle for ou de code en général comme nous allons voir dans la suite de ce livre.

Exemple :

Commençons par déclarer une variable nommée « myName » en utilisant le mot-clé **var** à l'intérieur d'un bloc délimité par des accolades, puis essayons d'accéder à cette variable en dehors de ce bloc.

```
{
    var myName = "Karine";
}

alert(myName);
```

Lorsque vous exécutez ce code, vous obtiendrez le résultat suivant :



Explication :

Ici vous déclarez une variable appelée **myName** en utilisant le mot-clé **var**. Bien que cette variable soit déclarée à l'intérieur d'un bloc, **var** a une portée **fonctionnelle** ou **globale**. Ce qui signifie que la variable myName n'est pas restreinte à ce bloc, mais peut être accédée en dehors du bloc car **var** n'a pas de portée de bloc (block scope).

Alors, notez ici que contrairement à **let**, la variable déclarée par **var** n'a pas la portée de bloc. La variable « myName » est déclarée ici à l'intérieur d'un bloc de code, cependant, elle est toujours accessible en dehors de ce bloc. Le fonctionnement du code est simple : la variable « myName » est définie avec la valeur « Karine » ici à l'intérieur du bloc mais elle reste accessible en dehors de ce bloc. Ainsi, en utilisant

alert(myName) à l'extérieur du bloc, vous obtenez « Karine ».

Remplaçons maintenant, dans le code précédent, la mot clé « **var** » par « **let** »

```
    {
        let myName = "Karine";
    }

     alert(myName);   // Erreur : `name` n'est pas défini en
dehors du bloc
```

Si vous exécutez ce code vous obtiendrez une erreur :

Vous remarquez, dans ce cas l'exécution de ce code génère une erreur. En effet la variable myName est déclarée par le mot clé let et par conséquent cette variable n'est pas accessible à l'extérieur du bloc où elle est définie.

En résumé :
La déclaration par var : la variable est accessible en dehors du bloc où elle a été déclarée.
La déclaration par let (et **const**) : la variable est limitée au bloc où elle est déclarée, et ne peut pas être accédée en dehors de ce bloc.

Re-déclaration dans le même scope
var : Permet de redéclarer une variable dans le même bloc ou la même fonction sans générer d'erreur.

let : Ne permet pas de redéclarer une variable dans le même bloc. Si tu essaies de redéclarer une variable avec let dans le même bloc, cela génère une erreur.

Exemple de redéclaration avec le mot clé « var »:

```
var myName = 'Alice';
var myName = 'Bob';   // Pas d'erreur, `myName` peut être
redéclarée avec `var`
```

Dans ce cas nous avons redéclaré la variable myName, le code

s'exécute correctement sans aucune erreur.

Exemple de redéclaration avec le mot clé « let »:

```
let age = 25;
let age = 30;  // Erreur : ne peut pas redéclarer une variable
avec `let`
```

Dans la première ligne, une variable **age** est déclarée avec **let** et initialisée à 25. Cette variable est désormais accessible dans le bloc de code où elle a déclarée, par exemple, dans la fonction, la boucle etc. Cependant, lorsque l'on tente de redéclarer **age**, JavaScript génère une erreur de référence. Néanmoins, let ne permet pas de redéclaration de variables dans le même bloc ou le même contexte. Il est fait pour empêcher les conflits de noms et les problèmes associés à la redéfinition accidentelle de variables. C'est pourquoi il est important de garder à l'esprit la portée des variables quand elles sont déclarées par **let**.

Lorsque vous exécutez ce code, vous obtiendrez l'erreur suivante :

```
Uncaught SyntaxError: Identifier 'age' has already been declared
```

Cela signifie que vous essayez redéclarer une variable déjà déclarée avec **let** dans le même contexte. Cette erreur de syntaxe implique qu'elle y'a une syntaxe incorrecte pour le compilateur JavaScript. Donc, cette règle vous empêche la redéclaration accidentelle de variables, qui est commune dans les anciens codes avec **var**.

Par conséquent, quand vous utilisez **let** sous le mode « **strict** », vous gérez plus facilement la portée des variables et vous minorez les erreurs dues à des collisions de noms de variables.

Quant au mot clé **var** permet de redéclarer une variable dans le même bloc, mais il est plus judicieux de ne pas le faire : d'un côté, **let** ne l'autorise pas, d'un autre, vous rencontrez des comportements inattendus liées à réaffectation après sa déclaration.

Pour résumer, **let** ne vous autorise pas à redéclarer une variable dans le même bloc, alors JavaScript vous envoyait une erreur de syntaxe si vous essayez de redéclarer une variable dans le même contexte. Si vous voulez contraindre la portée de votre variable à un bloc spécifique, utilisez **let**, et si vous ne souhaitez pas la réaffectation après sa déclaration, préférez **const**.

Afficher les résultats dans la console

Depuis le début, nous utilisons la fonction **alert()** pour afficher les résultats dans des boîtes de dialogue. Même si cela peut fonctionner, il n'est généralement pas si pratique ou efficace, car afficher des résultats dans une autre fenêtre nécessite de fermer cette fenêtre pour continuer. En outre, elle peut arrêter le navigateur ou même la page. En réalité, dans la plupart des cas, nous aimerions voir les résultats apparaître directement dans la console plutôt que dans une boite d'alerte. De cette façon, vous pouvez continuer à lire le code et à voir les résultats. D'autre part, cela vous permet de suivre et de tester vos scripts sans interrompre l'expérience utilisateur, en envoyant des fenêtres d'alerte non demandées. La console est une méthode courante et puissante pour déboguer votre code. Vous pouvez également visualiser les informations importantes en temps réel. Par exemple, pour rechercher des mises en garde et des erreurs. Pour afficher les résultats dans la console en javascript, nous utilisons la fonction **console.log().** Cela permet d'envoyer des messages ou des résultats à la console du navigateur qui est un outil de développement très utile pour déboguer et suivre l'exécution de votre code.
Voici comment cela fonctionne :

Utilisation de console.log()

La fonction **console.log()** permet d'afficher dans la console du navigateur tout ce que vous lui passez comme argument. Cela peut être des nombres, des chaînes de caractères, des variables, ou même des objets qui seront étudiés dans les chapitres suivants.

Exemples simples :

Exemple 1 :

```
    let number = 10;
     console.log("Le nombre est : " + number); // Affiche "Le
nombre est : 10"
```

Exemple 2 :

```
    let city = "Paris";
    console.log(city); // Affiche "Paris"
```

Où voir les résultats dans la console ?
Pour voir ce qui est affiché par **console.log()**, vous devez ouvrir la console de développement de votre navigateur :

Si vous utilisez Chrome (ou d'autres navigateurs basés sur Chromium par exemple Edge) :

- Cliquez droit sur la page web et cliquez sur Inspecter :
- Ensuite cliquer sur l'onglet Console,
- Vous verrez ici tous les messages et les résultats de console.log().

Si vous utilisez Firefox :

- Cliquez droit sur la page web et cliquez sur Inspecter l'élément,
- Puis cliquer sur l'onglet Console,

Si vous avez exécuté l'exemple 1, dans la console de votre navigateur vous trouverez le message " Le nombre est : 10 ".
Si vous avez exécuté l'exemple 2, dans la console de votre navigateur vous trouverez le message " Paris ". Démonstration avec la console.

En résumé, la fonction **console.log()** est essentielle pour tester et visualiser les résultats de votre code dans la console du navigateur. Elle vous aide non seulement à afficher des messages, mais aussi à suivre et déboguer l'exécution de votre programme.

Conclusion

Nous avons exploré les fondamentaux des variables, constantes et types de données dans ce chapitre de JavaScript. Vous avez appris comment déclarer des variables à l'aide de var, let, et des constantes à l'aide de const, ainsi que leurs modes de fonctionnement et d'utilisation. Les types de données jouent un rôle essentiel dans la modification et le traitement des informations dans vos programmes, tels que les nombres, les chaînes de caractères, les booléens et des types spéciaux comme undefined et null, en plus de l'utilisation de l'opérateur typeof pour vérifier et manipuler efficacement les types de données.
Nous avons également examiné les différences clés entre 'var' et 'let' utilisés pour déclarer les variables. Ces différences joueront un rôle important dans la manipulation des données dans vos futurs programmes.
Nous pouvons déclarer des variables pour stocker des informations en utilisant 3 différentes expressions var, let et const. Soit :

- **let** est un nouveau mot clé pour déclarer les variables, qui est plus générale et plus spécifique du bloc que var.
- **var** est le premier moyen de déclaration des variables en JavaScript et il a été ignoré dans les versions relativement récentes de javaScript.
- **const** est similaire à **let** dans quelque dimensions,

cependant, une fois une valeur lui a été affectée, la variable déclarée avec cette expression ne peut pas être modifiée. Elle est utilisée lorsqu'il est nécessaire qu'une variable reste définie tout au long du programme.

Dans les prochains chapitres nous continuons à approfondir les opérations sur la manipulations des données ainsi que les structures conditionnelles qui nous permettent de prendre des décision selon les conditions définies.

Chapitre 3

Les opérateurs

Après avoir appris à maîtriser les bases des variables et les types de données, le moment est venu pour étudier la notion des opérateurs. En fait, les opérateurs sont des outils essentiels qui permettent de transformer, comparer et associer des valeurs en JavaScript. Les opérateurs sont au cœur de toutes les actions effectuées dans vos scripts, qu'il s'agisse de simples calculs mathématiques ou de comparaisons complexes.

Que vous souhaitiez additionner deux nombres, vérifier si une condition est remplie ou combiner des chaînes de caractères, les opérateurs sont vos alliés pour faire interagir vos variables et prendre des décisions dans vos programmes.

Dans ce chapitre, nous allons explorer l'univers des opérateurs, des bases comme l'addition et la soustraction aux plus avancés tels que les opérateurs de comparaison et logiques. Vous découvrirez comment JavaScript utilise ces opérateurs pour manipuler les données et comment les employer efficacement pour développer la dynamique et l'intelligence de vos programmes.

Préparez-vous à développer vos compétences en programmation en comprenant comment ces petits symboles peuvent faire une réelle différence!

Opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur les nombres. Ils permettent aussi d'ajouter, de soustraire, de multiplier ou de diviser des valeurs numériques. Ces opérateurs sont indispensables dans la plupart des langages de programmation y compris le JavaScript. En fait, ils sont essentiels pour réaliser des calculs et sont nécessaires pour manipuler des variables quantitatives.

Voici les principaux opérateurs mathématiques en JavaScript :

Addition (+) : L'opérateur + sert à additionner deux nombres ou à concaténer des chaînes de caractères. L'addition regroupe ou réunit des quantités.

Exemple :

```
let x = 10;
let y = 5;
let result = x + y; // result vaut 15
console.log(result);
```

Dans la console, vous obtiendrez 15 comme résultat de la somme de x et y.

Soustraction (-) : L'opérateur - permet de soustraire un nombre d'un autre.

Exemple :

```
let x = 10;
let y = 5;
let result = x - y; // result vaut 5

console.log(result);
```

Multiplication (*) : L'opérateur * permet de multiplier deux nombres.

Exemple :

```
let x = 10;
```

```
let y = 5;
let result = x * y; // result vaut 50

console.log(result);
```

Division (/) : L'opérateur / permet de diviser un nombre par un autre.

Exemple :

```
let x = 10;
let y = 5;
let result = x / y; // result vaut 2

console.log(result);
```

Modulo (%) :

Le résultat renvoyé par l'application de l'opérateur % est le reste d'une division. Il est souvent utilisé pour déterminer si un nombre est pair ou impair.

Exemple :

```
let x = 10;
let y = 3;
 let result = x % y; // result vaut 1, car 10 divisé par 3 donne un reste de 1

console.log(result);
```

Exponentiation () :**

L'opérateur de puissance (exponentiation), symbolisé par deux étoiles (**) permet de calculer une puissance. C'est une manière simple d'élever un nombre à une certaine puissance.

```
let x = 2;
let result = x ** 3; // result vaut 8 (2 à la puissance 3)
```

```
    console.log(result);
```

Le résultat affiché dans votre console sera 8 (2 à la puissance 3).

Incrémentation (++) :

L'opérateur ++ permet d'ajouter 1 à la valeur d'une variable.

Exemple 1 :

```
    let x = 10;
    x++; // x vaut maintenant 11

    console.log(x);
```

x++ utilise l'opérateur d'incrémentation symbolisé par deux symboles plus (++), il permet d'ajouter 1 à la valeur actuelle de **x**. Après cette opération, **x** prend la valeur de 11.
Cet opérateur est très pratique pour augmenter la valeur d'une variable sans avoir à écrire une opération complète comme x = x + 1.

L'exécution de la ligne de code : **console.log(x);** affiche dans la console de votre navigateur la nouvelle valeur de la variable x.

En résumé, ce code utilise l'opérateur d'incrémentation pour augmenter la valeur de x de 10 à 11.

Exemple 2 :

```
    let x = 5;
     let y = x++; // y prend la valeur actuelle de x (5), puis
x est incrémenté à 6
    console.log("la valeur de x est: " + x); // Affiche 6
    console.log("La valeur de y est: " + y); // Affiche 5
```

Si vous exécutez ce code, vous obtiendrez le résultat suivant :

```
la valeur de x est: 6
La valeur de y est: 5
```

Comment expliquer ce résultat ?

Dans cet exemple, l'opérateur d'incrémentation (++) est placé après la variable x, l'incrémentation se fait donc **après** son utilisation. La valeur initiale de x est utilisée dans l'expression, puis sa valeur est augmentée de 1. En résumé, la variable y reçoit d'abord la valeur actuelle de x (5), et seulement ensuite x passe à 6. L'incrémentation de la variable x n'est appliquée qu'après son emploi dans l'expression.

À l'inverse, lorsque l'opérateur d'incrémentation est mis **avant** la variable, l'augmentation intervient immédiatement, avant toute utilisation. C'est-à-dire que la valeur de x est augmentée de 1 en amont, et c'est cette nouvelle valeur qui est exploitée dans l'expression.

Voyons cet autre exemple :

```
    let x = 5;
     let y = ++x; // x est incrémenté à 6, puis la nouvelle
valeur de x est assignée à y

    console.log("la valeur de x est: " + x); // Affiche 6
    console.log("La valeur de y est: " + y); // Affiche 5
```

Lorsque vous exécutez ce code, vous aurez dans la console de votre navigateur, l'affichage suivant :

```
la valeur de x est: 6
La valeur de y est: 6
```

Résumé :

- **++x (incrémentation avant)** : La variable est incrémentée avant d'être utilisée dans l'expression. Le résultat donne la nouvelle valeur.
- **x++ (incrémentation après)** : La variable est incrémentée après avoir été utilisée dans l'expression. Le

résultat est l'ancienne valeur, et l'incrémentation a lieu après.

Décrémentation (--) :

Bien que l'opérateur de **décrémentation** (double signe moins : --) fonctionne de façon analogue à l'opérateur d'incrémentation (++), au lieu d'ajouter 1 à la variable, il en soustrait 1. Tout comme pour l'incrémentation, la place de l'opérateur de décrémentation par rapport à la variable influence l'ordre dans lequel la diminution s'effectue.

Lorsque l'opérateur de décrémentation précède la variable, celle-ci se voit retirer 1 immédiatement, avant même que sa nouvelle valeur ne soit employée dans l'expression. À l'inverse, quand il est placé après la variable, celle-ci est d'abord utilisée telle quelle, puis décrémentée de 1 ensuite.

Exemple :

```
    let x = 5;
     let y = --x; // x est décrémenté à 4, puis la nouvelle
valeur de x est assignée à y
    console.log(x); // Affiche 4
    console.log(y); // Affiche 4
```

Dans cet exemple, le symbole de décrémentation est placé avant la variable x et par conséquent, x est décrémenté avant d'être assigné à y. Ainsi, x et y valent tous les deux 4 après l'opération.

À l'inverse, si l'opérateur de décrémentation est placé après la variable, la variable est utilisée dans l'expression avec sa valeur actuelle, puis décrémentée de 1 dans la suite du code.

Exemple :

```
    let x = 5;
     let y = x--; // y prend la valeur actuelle de x (5), puis
x est décrémenté à 4
    console.log(x); // Affiche 4
    console.log(y); // Affiche 5
```

Dans cet exemple, initialement, la valeur de la variable x était définie à 5. Lors du premier calcul, elle a été utilisée dans son état avant d'être décrémentée à 4. Par conséquent, y a pris la valeur de x, soit 5, alors que la valeur de x est devenue 4

Résumé :

- **--x (décrémentation avant)** : La variable est soustraite de 1 avant son utilisation dans l'expression. C'est donc sa nouvelle valeur qui est employée.

- **x-- (décrémentation après)** : Ici, c'est la valeur d'origine de la variable qui est retenue pour le calcul, puis celle-ci est décrémentée de 1 seulement après.

Ordre des opérations

Bien que la compréhension des opérateurs soit importante, l'ordre de priorité peut parfois être négligée. En effet, les symboles mathématiques suivent une hiérarchie précise qu'il ne faut pas ignorer, communément appelé ordonnancement des opérations. Ainsi, la multiplication et la division sont réalisées avant l'addition et la soustraction, à moins que des parenthèses ne changent cette séquence. Cependant, un rappel méticuleux des règles garantira des calculs corrects.

Exemple :

```
let result = 10 + 5 * 2; // result vaut 20, car 5 * 2 est évalué avant l'addition
```

Il existe un ordre de priorité dans le traitement des expressions mathématiques en programmation. Selon les règles de priorité, la multiplication et la division sont effectuées avant l'addition et la soustraction.

Dans cet exemple numérique, le calcul 5×2 est réalisé en premier lieu car la multiplication prend place devant l'addition. Le résultat obtenu est donc 10 pour l'opération 5×2.

Résultat final :

Ensuite, l'addition $10 + 10$ est effectuée, ce qui donne le résultat final 20.

Donc, la variable result prend la valeur 20 après l'évaluation complète de l'expression.

Pour forcer l'addition à s'effectuer en premier, utilisez des parenthèses :

```
let result = (10 + 5) * 2; // result vaut 30, car 10 + 5 est évalué en premier
```

Voici un tableau qui résume les opérations des opérateurs arithmétiques :

Opérateur	Opération associée
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste d'une division euclidienne)
**	Élévation à la puissance d'un nombre par un autre (ES2016)
++	Incrémentation de 1 (Ajout de 1)
--	Décrémentation de 1

Opérateur d'affectation

En JavaScript, l'opérateur d'affectation de base est le symbole `=`. Il constitue l'un des opérateurs les plus utilisés en programmation et permet d'assigner une valeur à une variable..

Exemple :

```
let x = 10;
```

Ici, l'opérateur `=` affecte la valeur **10** à la variable `x`. Ce n'est pas un signe d'égalité comme en mathématiques, mais bien un symbole pour **assigner** une valeur à une variable.

Combinaison des opérateurs arithmétiques avec l'opérateur d'affectation

Les opérateurs combinés sont couramment utilisés en JavaScript, à l'instar de la plupart des langages de programmation, pour effectuer une opération mathématique et attribuer le résultat à une variable en une seule étape. Cela permet de condenser et clarifier le code.

En voici quelques exemples typiques :

L'addition attribuée (+=) : Cet opérateur ajoute la valeur située à droite du signe à la variable, puis réattribue la somme à cette même variable. De façon analogue, la soustraction avec affectation (-=), la multiplication avec affectation (*=) et la division avec affectation (/=) réalisent respectivement une opération de soustraction, multiplication ou division suivie d'une réaffectation du résultat.

Exemple 1 :

```
let x = 10;
x += 5; // équivaut à x = x + 5, donc x vaut maintenant 15
```

Le résultat de la ligne de code **x += 5** est équivalent à celui de x = x + 5. C'est-à-dire le calcul de x=5 est fait en premier et le résultat obtenu est affecté à la variable x.
Donc, plutôt que d'écrire x = x+5, vous pouvez simplement écrire x +=5 et vous obtiendrez le même résultat. En fait, l'utilisation des opérateurs combinés présente plusieurs avantages :

- Clarté : Le code est plus compact et lisible.
- Efficacité : Vous réalisez l'opération et l'affectation en une seule étape.

Exemple 2 :

Modulo avec affectation (%=) : Cet opérateur divise la variable par la valeur à droite de l'opérateur et affecte le reste à la variable.

```
let x = 10;
x %= 3; // équivaut à x = x % 3, donc x vaut maintenant 1
```

Dans cet exemple, l'opération **x % 3** (soit 10 % 3) est effectuée en premier. Le reste de la division de 10 par 3 est 1, et c'est cette valeur qui est ensuite affectée à la variable x. Ainsi, après l'exécution, x prend la valeur 1.

Comme le montre le tableau ci-dessous, le même principe reste vrai pour la combinaison de tous les opérateurs arithmétiques et l'opérateur d'affectation (=).

Opération	Opération équivalente
x += y	x = x + y

```
x -= y  x = x - y
x *= y  x = x * y
x /= y  x = x / y
x %= y  x = x % y
x **= y  x = x ** y
```

Les opérateurs de comparaison

Les opérateurs de comparaison en JavaScript servent à évaluer la correspondance entre deux valeurs. Ces opérateurs permettent de déterminer si des valeurs sont équivalentes, dissemblables, ou si l'une surpasse ou reste en deçà de l'autre. Le résultat d'une comparaison est immanquablement un booléen (vrai ou faux), qui révèle si la condition estimée est vraie ou erronée.

Égalité stricte (===)
L'opérateur d'égalité stricte vérifie si deux valeurs coïncident totalement tant au niveau de la valeur que du type.

Exemple :

```
    let a = 5;
    let b = "5";
     console.log(a === b); // Affiche false, car '5' (chaîne) n'est pas égal à 5 (nombre)
```

Explication :

- a est un nombre, avec la valeur 5.
- b est une chaîne de caractères, avec la valeur "5".
- L'opérateur === vérifie à la fois que la valeur et le type sont identiques. Ici, bien que les valeurs soient similaires, leurs types sont différents (nombre vs chaîne). Le résultat est donc **false** (faux).

Égalité faible (==)
L'égalité faible compare les valeurs après avoir converti les types.

Exemple :

```
    let a = 5;
    let b = "5";
```

```
    console.log(a == b); // Affiche true, car '5' (chaîne) est
converti en 5 (nombre) avant la comparaison
```

Explication :

- L'opérateur == essaie de convertir les deux valeurs au même type avant de les comparer.
- Dans cet exemple, b (une chaîne) est converti en un nombre, donc les deux valeurs deviennent 5 et sont considérées comme égales. Le résultat est donc true (vrai).

Différence stricte (!==)
La différence stricte vérifie si deux valeurs sont différentes en termes de type ou de valeur.

Exemple :

```
    let a = 5;
    let b = "5";
     console.log(a !== b); // Affiche true, car '5' (chaîne)
est différent de 5 (nombre)
```

Explication :

- Ici, même si les valeurs semblent similaires, leurs types sont différents (nombre vs chaîne).
- L'opérateur **!==** vérifie à la fois la valeur et le type. Comme les types sont différents, le résultat est **true** (ils sont bien différents).

Différence faible (!=)
La différence faible compare deux valeurs après avoir converti leurs types.

Exemple :

```
    let a = 5;
    let b = "5";
        console.log(a != b); // Affiche false, car après
conversion, 5 et '5' sont considérés comme égaux
```

Explication :

- Comme pour l'égalité faible, l'opérateur **!=** essaie de convertir les deux valeurs au même type avant la comparaison.
- Après conversion, b devient un nombre (5), et les deux valeurs sont donc considérées comme égales. Le résultat est donc false (elles ne sont pas différentes après conversion).

Supérieur à (>) et supérieur ou égal à (>=)
Ces opérateurs comparent si une valeur est supérieure ou égale à une autre.

Exemple 1 :

```
    let a = 10;
    let b = 5;
     console.log(a > b); // Affiche true, car 10 est supérieur
à 5
```

Explication :

Ici, l'opérateur > compare 10 et 5. Comme 10 est plus grand que 5, le résultat est true.

Exemple 2 :

```
   let a = 10;
     console.log(a >= 10); // Affiche true, car 10 est égal à
10
```

Explication :

L'opérateur >= compare si a est supérieur ou égal à 10. Comme a vaut exactement 10, la condition est vraie. Le résultat est donc true.

Inférieur à (<) et inférieur ou égal à (<=)
Ces opérateurs comparent si une valeur est inférieure ou égale à une autre.

Exemple 1 :

```
    let a = 10;
```

```
    let b = 5;
    console.log(b < a); // Affiche true, car 5 est inférieur à
10
```

Explication :

L'opérateur < compare 5 et 10. Comme 5 est bien inférieur à 10, le résultat est true.

Comparaison de chaînes de caractères
Les chaînes de caractères sont comparées selon l'ordre alphabétique basé sur les valeurs Unicode.

Exemple :

```
    console.log("apple" > "banana"); // Affiche false, car 'a'
vient avant 'b'
```

Explication :

- En JavaScript, les chaînes de caractères sont comparées en utilisant l'ordre des lettres dans le codage Unicode. Étant donné que "apple" commence par la lettre a et "banana" par b, et que **a** précède **b** dans l'alphabet, "apple" est considéré comme inférieur à "banana". Le résultat est donc faux.

Ces opérateurs de comparaison sont essentiels pour prendre des décisions dans un programme en JavaScript, permettant d'évaluer des expressions et d'agir en conséquence du résultat. Le tableau ci-dessous résume brièvement ces opérateurs :

Définition	Opérateur
Inférieur strictement	<
Supérieur strictement	>
Inférieur ou égale	<=
Supérieur ou égale	>=
Pour tester l'égalité en termes de valeurs	==
Pour tester l'égalité en termes de valeurs et de types	===
Pour tester la différence en termes de valeurs	!=
Pour tester la différence en valeurs ou en types	!==

Opérateur de concaténation

En JavaScript, l'opérateur de concaténation est utilisé pour assembler ou coller ensemble plusieurs suites de chaînes de caractères. Ce symbole +, représentant cet opérateur, sert également au calcul d'addition. Cependant, lorsqu'il est utilisé avec des chaînes textuelles, son rôle est de les concaténer plutôt que de les additionner.

Utilisation de l'opérateur + pour concaténer
L'opérateur + permet de combiner deux ou plusieurs chaînes en une seule.

Exemple :

```
let firstName = "Jean";
let lastName = "Dupont";
let fullName = firstName + " " + lastName;
console.log(fullName); // Affiche "Jean Dupont"
```

Explication :

- Dans cet exemple, les variables firstName et lastName sont des chaînes de caractères. L'opérateur + est utilisé pour concaténer (assembler) ces deux chaînes en ajoutant également un espace " " entre elles.
- Le résultat final est "Jean Dupont", qui est affiché dans la console.

Concaténation de chaînes avec des valeurs numériques
Lorsque vous concaténez une chaîne de caractères avec une valeur numérique, JavaScript convertit automatiquement la valeur numérique en chaîne de caractères, puis les concatène.

Exemple :

```
let age = 25;
let message = "J'ai " + age + " ans.";
console.log(message); // Affiche "J'ai 25 ans."
```

Explication :

Ici, la variable age est un nombre, mais lorsqu'elle est concaténée avec la chaîne "J'ai " et " ans.", JavaScript convertit automatiquement age en une chaîne de caractères.
Le résultat final est "J'ai 25 ans.".

Utilisation des backticks (template literals) pour concaténer

Bien que l'opérateur + soit couramment utilisé pour la concaténation, JavaScript offre une autre méthode plus moderne et lisible : Cette méthode consiste à utiliser les **backticks** (`) ou gabarits de chaînes (template literals). Ils permettent d'insérer des variables directement dans une chaîne de caractères à l'aide de la syntaxe ${}.

Exemple :

```
let firstName = "Jean";
let lastName = "Dupont";
let fullName = `${firstName} ${lastName}`;
console.log(fullName); // Affiche "Jean Dupont"
```

Explication :

- Dans cet exemple, les backticks (`) remplacent les guillemets, et les variables firstName et lastName sont insérées directement dans la chaîne à l'aide de ${}. Cela rend la concaténation plus simple et lisible.
- Le résultat final est identique à celui obtenu avec l'opérateur +.

Concatenation de plusieurs chaînes
Vous pouvez concaténer plusieurs chaînes ou variables en les combinant les unes après les autres.

Exemple :

```
let greeting = "Bonjour";
let name = "Alice";
let message = greeting + ", " + name + "!";
console.log(message); // Affiche "Bonjour, Alice!"
```

Explication :

Ici, trois éléments sont concaténés dans une chaîne de caractères : la variable « greeting », une chaîne de caractères, la virgule ", " et la variable « name ». Le résultat final "Bonjour, Alice!" est affiché dans la console.

En résumé, l'opérateur de concaténation (+) est essentiel pour assembler des chaînes de caractères en JavaScript. Qu'il s'agisse de combiner des mots, d'ajouter des espaces ou d'associer des valeurs numériques à des chaînes, il offre une grande flexibilité. Avec l'introduction des backticks (`), la concaténation est devenue encore plus simple et lisible, notamment lorsque plusieurs variables interviennent dans une chaîne complexe.

Conclusion

Dans ce chapitre, nous avons exploré les différents opérateurs qui constituent les outils essentiels pour manipuler et interagir avec les données en JavaScript. Que ce soit les opérateurs arithmétiques permettant d'effectuer des calculs, les opérateurs de comparaison aidant à prendre des décisions logiques ou les opérateurs de concaténation facilitant la manipulation des chaînes de caractères, chacun de ces types d'opérateurs joue un rôle clé dans le développement de programmes dynamiques et performants.

Nous nous sommes également penchés sur les opérateurs d'affectation et leurs combinaisons avec des opérations arithmétiques, offrant une manière concise de mettre à jour les valeurs des variables.

En maîtrisant ces opérateurs, vous êtes désormais capable d'écrire des scripts JavaScript plus complexes et puissants, tout en améliorant la clarté et l'efficacité de votre code.

Les opérateurs sont omniprésents en programmation. Une bonne utilisation de ceux-ci vous permettra de résoudre des problèmes de manière plus élégante et d'optimiser vos futurs projets. Dans les chapitres suivants, nous continuerons à développer vos compétences en JavaScript en approfondissant les structures de contrôle, les fonctions et bien d'autres concepts.

Chapitre 4

Les structures conditionnelles

Tous les langages de programmation comprennent que pendant l'exécution d'un programme, il est nécessaire de prendre des décisions en fonction de certaines conditions. Par conséquent, il existe des structures conditionnelles qui régulent cela. En d'autres termes, les structures conditionnelles permettent de contrôler quel bloc de code devrait être exécuté à un moment donné pendant l'exécution d'un script en fonction de différentes conditions.

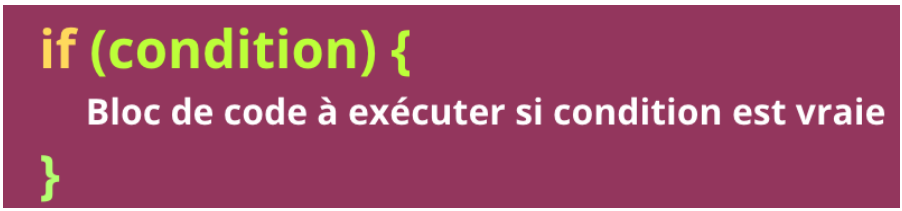
Plus précisément, en JavaScript, les structures conditionnelles sont un outil qui vous permet de rendre vos programmes dynamiques et interactifs. Ils vous permettent d'écrire des programmes qui s'adapteront à différentes situations. Sans structures conditionnelles, les programmes seront linéaires, statiques et ne réagissent pas selon les besoins.

Dans ce chapitre, nous explorons plusieurs structures conditionnelles JavaScript, à savoir if, else, else…if. switch et les opérateurs ternaires. Dans cet article, nous découvrirons des aspects tels que: Cela permettra de découvrir comment vous pouvez contrôler les programmes en fonction des divers états.

L'instruction if

Le premier cas des structures conditionnelles à savoir : l'instruction if permet d'exécuter un bloc de code dans le cas où la condition spécifiée est vraie. Mais, dans le cas contraire, le bolc de code sera tout simplement ignoré.

Syntaxe :



Exemple :

```
let age = 18;

if (age >= 18) {
    console.log("Vous êtes majeur");
}
```

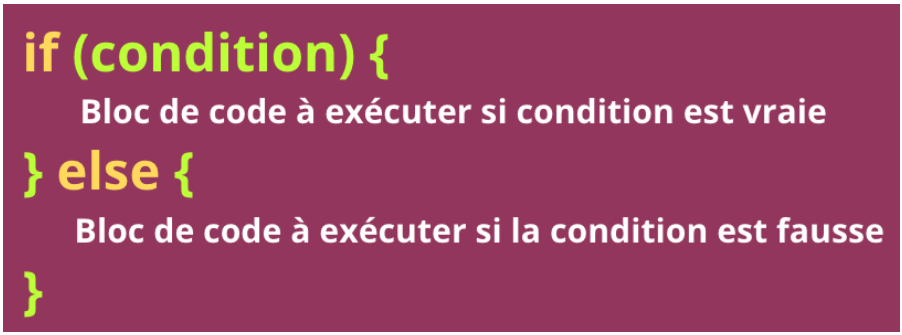
Explication :

Dans cet exemple, nous commençons par vérifier si la valeur de la variable « age » est supérieur ou égale à 18. Si c'est le cas le bloc de code entre les accolades sera exécuté et le message « Vous êtes majeur » sera affiché dans la console de votre navigateur. Mais si la age n'est pas supérieur à 18, le programme saute le bloc de code entre les accolades et aucun message n'est affiché.

L'instruction else

L'instruction **else** permet de spécifier le bloc de code à exécuter dans le cas où la condition spécifiée dans le **if** est fausse.

Syntaxe :



Exemple :

```
let age = 16;

if (age >= 18) {
    console.log("Vous êtes majeur");
} else {
    console.log("Vous êtes mineur");
}
```

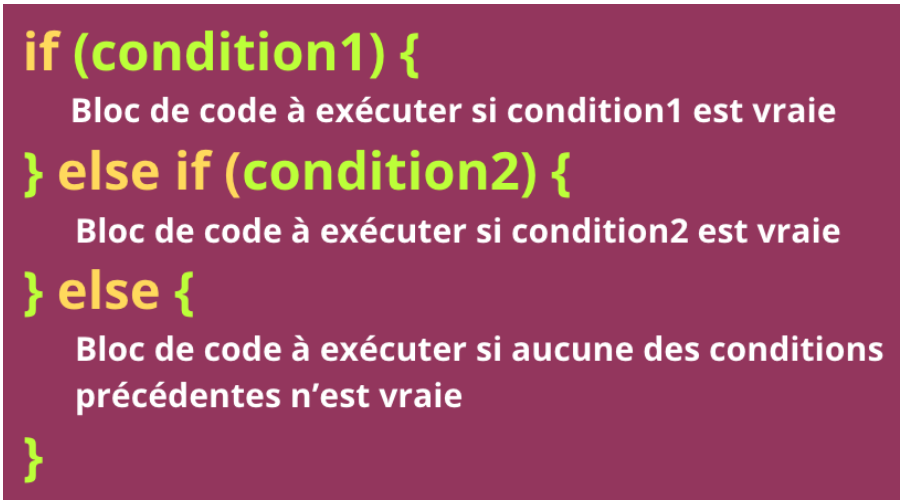
Explication :

Dans cet exemple, puisque « age » est égale 16, c'est-à-dire la condition age >= 18 est fausse et sa négation qui vraie (c'est-à-dire age est inferieur à 18) et par conséquent c'est le bloc de **else** qui sera exécuté. Pour cela, le message « Vous êtes mineur sera affiché dans la console de votre navigateur.

L'instruction else if

L'instruction else…if combine le else et le if. Elle permet d'exécuter un bloc de code dans le cas où la ou les conditions précédentes sont fausses et la condition spécifiée dans le else…if est vraie. On peut utiliser une succession de else…if selon nos besoins.

Syntaxe :



Exemple :

```
let age = 20;

if (age < 13) {
    console.log("Vous êtes un enfant");
} else if (age >= 13 && age < 18) {
    console.log("Vous êtes un adolescent");
} else {
    console.log("Vous êtes un adulte");
}
```

Si la première condition est vérifiée, c'est-à-dire si l'âge est inférieur à 13 ans, le message "Vous êtes un enfant" s'affichera. Dans le cas contraire, si l'âge est compris entre 13 et 17 ans inclus, le message "Vous êtes un adolescent" apparaitra. À défaut, c'est-à-dire si les deux conditions précédentes sont fausses et que l'âge est supérieur ou égal à 18 ans, le message "Vous êtes un adulte" sera affiché.

Dans cet exemple, la variable âge étant égale à 18, le dernier message sera bien affiché lors de l'exécution du code.

Quelques éléments clés à retenir : après un premier test if, on peut enchainer avec plusieurs else if successifs. Le bloc de code du premier else if dont la condition est vraie sera exécuté. En d'autres termes, une condition else if ne sera évaluée que si toutes les conditions précédentes (if et else if antérieurs) sont fausses, et que la condition

actuelle est vraie.

L'instruction switch :

Lorsqu'il y a de multiples tests à effectuer sur une même variable, on préférera utiliser l'instruction switch plutôt qu'une série de if/else imbriqués, afin d'améliorer la lisibilité et la compréhension du code.
Syntaxe :

```
switch (expression) {
  case valeur1:
    // Code à exécuter si expression vaut valeur1
    break;
  case valeur2:
    // Code à exécuter si expression vaut valeur2
    break;
  default:
    // Code à exécuter si aucune des valeurs ne correspond
}
```

```
let jour = "lundi";

switch (jour) {
    case "lundi":
        console.log("Aujourd'hui, c'est lundi");
        break;
    case "mardi":
        console.log("Aujourd'hui, c'est mardi");
        break;
    default:
        console.log("Je ne connais pas ce jour");
}
```

Puisque la valeur de la variable jour était "lundi", c'est donc le message "Aujourd'hui, c'est lundi" qui s'afficha dans la console de votre navigateur.

Les variables booléennes et leurs usages dans les conditions

Une variable booléenne ne peut prendre que deux états de vérité : soit vrai, soit faux. Ces états servent à déterminer si un bloc de code particulier doit ou non s'exécuter, car ils indiquent si une condition donnée est remplie ou non.

Les variables booléennes trouvent une application de choix dans les structures conditionnelles. Par exemple, vous pouvez tester directement une variable booléenne dans une condition if sans avoir à utiliser d'opérateurs de comparaison.

Exemple 1 : Test d'une variable booléenne

```
let estMajeur = true;

if (estMajeur) {
    console.log("Vous êtes majeur");
} else {
    console.log("Vous êtes mineur");
}
```

Explication :

- Dans cet exemple, la valeur logique de la variable "estMajeur" est vrai. L'instruction if vérifie si la valeur de cette variable est vraie (true). Puisqu'elle est vrai dans cet exemple, c'est le message "Vous êtes majeur" qui s'affichera.
- Si la variable estMajeur avait été initialisée à faux au lieu, c'est le message "Vous êtes mineur" qui se serait affiché.

Exemple 2 : Modifier la valeur d'une variable booléenne

Le programme peut aussi changer la valeur d'une variable booléenne en fonction des conditions rencontrées pendant son déroulement.

```
let aPermission = false;
let age = 17;

if (age >= 18) {
    aPermission = true;
}

if (aPermission) {
```

```
        console.log("Accès autorisé");
    } else {
        console.log("Accès refusé");
    }
```

Explication :

Ici, la variable `aPermission` est initialement définie sur false. Si l'âge est supérieur ou égal à 18 ans, sa valeur est modifiée à true.

Ensuite, une deuxième condition vérifie la valeur de `aPermission`. Si elle vaut true, l'accès est permis, sinon il est refusé.

En résumé, les variables booléennes jouent un rôle fondamental dans les structures conditionnelles en JavaScript. Elles permettent de prendre des décisions de manière claire et simple, et leur manipulation est essentielle pour rendre vos programmes dynamiques et réactifs. Que ce soit pour gérer des autorisations, valider des conditions, ou contrôler le flux d'exécution d'un programme, les booléens sont omniprésents dans le développement en JavaScript.

L'imbrication des conditions

En JavaScript, il est souvent nécessaire de vérifier plusieurs conditions avant d'exécuter un bloc de code. Quand les cas à gérer sont complexes, on peut imbriquer des structures conditionnelles, c'est-à-dire placer une condition à l'intérieur d'une autre. Cela permet de vérifier plusieurs niveaux de conditions et de répondre à des scénarios précis.

Qu'est-ce que l'imbrication des conditions ?
L'imbrication des conditions consiste à inclure une structure if ou else if à l'intérieur d'une autre déjà présente. De cette façon, on peut évaluer plusieurs critères avant de prendre une décision et de déterminer quel fragment de code exécuter. Cette technique s'avère pratique lorsque les règles à appliquer sont nombreuses et précises.

Exemple simple d'imbrication :

```
    let age = 20;
    let permis = true;

    if (age >= 18) {
        if (permis) {
                console.log("Vous êtes majeur et vous avez un
permis de conduire.");
        } else {
               console.log("Vous êtes majeur, mais vous n'avez
pas de permis.");
        }
    } else {
        console.log("Vous êtes mineur.");
    }
```

Explication :

- Le premier if vérifie si l'âge de la personne est supérieur ou égal à 18 ans.
- Si cette condition est vraie, une seconde condition imbriquée teste ensuite si l'individu est titulaire d'un permis de conduire.
- Cependant, si l'âge est inférieur à 18 ans, le bloc else final sera alors exécuté pour spécifier que l'intéressé est

mineur.

- Grâce à cette imbrication de conditions, il est possible de gérer des niveaux multiples de décisions au sein du programme.

En exécutant ce code, la console de votre navigateur affichera les informations suivantes :

```
Vous êtes un adulte
```

Exemple plus complexe d'imbrication :

```
    let score = 85;
    let age = 16;

    if (score >= 80) {
        console.log("Excellent résultat.");
        if (age < 18) {
              console.log("Et en plus, vous êtes encore jeune
!");
        } else {
                  console.log("Un très bon résultat pour un
adulte.");
        }
    } else if (score >= 50) {
        console.log("Résultat moyen, peut mieux faire.");
    } else {
        console.log("Échec.");
    }
```

Explication :

- Si le nombre de points (score) était supérieur ou égal à 80, un message de félicitations serait affiché sur la console. On vérifierait ensuite l'âge de la personne à l'aide d'une condition imbriquée. Si la personne était mineure, un message supplémentaire s'afficherait. Dans le cas contraire, c'est-à-dire si la personne était majeure, un autre message serait destiné aux adultes.

- Entre 50 et 80 points, on aurait affiché un message encourageant l'utilisateur à continuer ses efforts.

- En deçà de 50 points, le message indiquant l'échec serait apparu, signifiant que les résultats n'étaient pas suffisants.

En exécutant ce code, on obtiendrait des résultats différents selon les valeurs des variables de points et d'âge.

```
Excellent résultat.
Et en plus, vous êtes encore jeune !
```

Points à retenir

- L'imbrication des conditions peut devenir complexe à lire et à comprendre si elle est trop profonde. Il est recommandé de garder l'imbrication aussi simple et claire que possible.
- Pour éviter les imbrications trop complexes, vous pouvez parfois utiliser des opérateurs logiques comme ET (&&) et OU (||) pour combiner plusieurs conditions en une seule phrase.

Les opérateurs logiques

Les opérateurs logiques en JavaScript sont utilisés pour **associer** ou **inverser** plusieurs conditions dans les structures de contrôle, comme les instructions if et else…if. Ils permettent de tester plusieurs conditions simultanément et d'exécuter du code en fonction de la logique souhaitée.

Les trois principaux opérateurs logiques sont :

ET (&&) :

Cet opérateur vérifie si toutes les conditions sont vraies. Si toutes les conditions sont évaluées à vrai, l'ensemble de l'expression est vrai. Si l'une des conditions est fausse, l'expression le sera aussi.

OU (||) :

Cet opérateur vérifie si au moins une des conditions est vraie. Si l'une des conditions est évaluée à vraie, l'expression le sera. L'ensemble de l'expression ne sera fausse que si toutes les conditions le sont.

NON (!) :

Cet opérateur inverse la valeur d'une condition. Si la condition est vraie, elle devient fausse, et vice versa.

Voici un tableau pour résumer les définitions des opérateurs logiques :

Symbole	Signification
AND	Renvoie true si toutes les comparaisons sont vraies (true)
OR	Renvoie true si au moins l'une des comparaisons est vraie (true)
Négation	Renvoie si la proposition est fausse et vice versa

Les conditions multiples

En JavaScript, il est souvent nécessaire de prendre des décisions complexes basées sur plusieurs critères importants. Pour réaliser cela, nous utilisons des opérateurs logiques qui permettent de combiner plusieurs conditions complexes. Cette section va explorer en profondeur les **opérateurs logiques**, leur utilisation stratégique, et comment ils permettent de créer habilement des conditions multiples.

Exemple 1 :

```
    let a = 100;
    let b = 32;
    if (a > 20 && b < 200){
            console.log("la condition est vraie car les deux conditions sont vraies !") ;
    }else{
           console.log("la condition est fausse car au moins l'une des deux conditions est fausse !");
    }
```

Explication :
Dans cet exemple élaboré, nous utilisons une instruction **if** pour évaluer une condition complexe. La condition se compose de deux sous-conditions liées par l'opérateur logique && (ET). Cela signifie que les deux sous conditions doivent être vraies pour que l'ensemble de la condition complexe soit considéré comme vraie.

- Première condition : a > 20 vérifie si 100 est supérieur à 20. Cette condition est vraie sans équivoque.
- Deuxième condition : b < 200 vérifie si 32 est inférieur à 200. Cette condition est également vraie clairement.

Comme les deux conditions sont vraies sans ambiguïté, l'ensemble de l'expression : a > 20 && b < 200 est vraie indubitablement. Par conséquent, lors de l'exécution de ce code, c'est le message spécifié dans le **if** qui sera affiché dans la console de manière détaillée.
Exemple 2 :

```
    let a = 10;
    let b = 32;
    if (a > 20 || b < 200){
        console.log("la condition est vraie car au moins l'une des conditions est vraie!") ;
```

```
    }else{
            console.log("la condition est fausse car les deux
conditions sont fausse !");
    }
```

Explication

Dans cet extrait de code, nous utilisons une instruction conditionnelle **if** pour évaluer une situation problématique intéressante. Cette circonstance problématique consiste en deux sous-conditions distinctes reliées par l'opérateur logique || (OU). Cela signifie que l'une ou l'autre des deux sous-conditions doit être vraie pour que l'ensemble de la circonstance conditionnelle soit considérée comme vraie.

- Premièrement, la sous-condition initiale : a supérieur à 20 vérifie si le chiffre 10 est plus grand que 20. Cette sous-condition est fausse.
- Deuxièmement, la seconde sous-condition : b inférieur à 200 vérifie si le nombre 32 est plus petit que 200. Cette sous-condition est vraie.

Étant donné qu'au moins une des sous-conditions est vraie (la seconde), l'intégralité de l'expression : « a supérieur à 20 || b inférieur à 200 » est vraie. Par conséquent, le message à l'intérieur du bloc de if est affiché.

Exemple 3 :

```
    let a = 100;
    let b = 32;

    if (!(a > 20)){
        console.log("la condition est vraie!") ;
    }else{
        console.log("la condition est fausse! !");
    }
```

Explication :

Dans cet exemple, nous avons une condition qui utilise l'opérateur logique de négation (!), qui inverse la valeur d'une expression.

- Premièrement, nous évaluons la condition a > 20, qui vérifie si la variable a est supérieure à 20. Cette condition est vraie car la valeur de la variable a est 100.

- Ensuite, l'opérateur de négation (!) inverse le résultat de cette condition. Donc : **!(a > 20)** devient faux parce que a > 20 est vrai.

Par conséquent, la condition **if (!(a > 20))** est fausse.

Conditions ternaires

Une façon concise d'écrire des conditions simples consiste à utiliser l'opérateur ternaire. Cela permet de formuler des conditions sur une seule ligne.

Syntaxe :

```
condition ? valeurSiVrai : valeurSiFaux;
```

Exemple :

```
    let age = 18;
     let message = (age >= 18) ? "Vous êtes majeur" : "Vous
êtes mineur";
    console.log(message); // Affiche "Vous êtes majeur"
```

Explication :

Considérons la ligne de code :

```
let message = (age >= 18) ? "Vous êtes majeur" : "Vous êtes
mineur";
```

Que dirait-on de laisser la variable « message » prendre une valeur basée sur l'âge de l'utilisateur ? Ici, **let message** assignera habilement soit "Vous êtes majeur" soit "Vous êtes mineur" en fonction de la condition, spécifiée entre parenthèses, sur l'âge.

Si l'âge se révèle être 18 ans ou plus, la variable message s'illuminera de la phrase "Vous êtes majeur". Cependant, si l'âge s'avère être moins de 18, la variable message prendra plutôt de la phrase "Vous êtes mineur".

Grâce à cet opérateur ternaire concis, nous pouvons évaluer la condition en une seule ligne et déterminer le contenu de message selon le résultat, le tout d'un seul tenant. C'est une manière éloquente de condenser ce qui autrement nécessiterait des lignes de code supplémentaires.

Bien sûr, cette ligne unique remplace le code alternatif suivant :

```
    let message;
```

```
if (age >= 18) {
    message = "Vous êtes majeur";
} else {
    message = "Vous êtes mineur";
}
```

En revanche, notez que ce code est plus explicite, surtout pour les personnes qui ne sont pas familières avec l'opérateur ternaire, mais il est plus long.

Une séquence d'opérateurs et l'opérateur ? (Point d'interrogation)

Imaginons que nous souhaitons afficher un message personnalisé en fonction de l'âge d'une personne. Plusieurs approches sont possibles pour y parvenir. La méthode traditionnelle consiste à utiliser une cascade d'instructions if/else if, comme illustré ci-dessous:

```
let age = 65;
let message;

if (age < 3) {
    message = 'Coucou mon bébé';
} else if (age < 18) {
    message = 'Vous n\'avez pas le droit de voter !';
} else if (age < 30) {
    message = 'Vous êtes encore jeune !';
} else if (age < 62) {
    message = 'Vous êtes encore actif !';
} else {
    message = 'Vous êtes certainement en retraite !';
}

console.log(message);
```

Explication :

- Une variable âge est déclarée et initialisée à 65 ans.
- Une variable message est définie pour stocker le texte approprié selon l'âge.
- Chaque condition **if, else...if** vérifie une tranche d'âge et

assigne le message adéquat.
- La première instruction **if** analyse si l'âge est inférieur à 3 ans, les **else if** suivent pour les autres tranches d'âge.
- Enfin, le message correspondant à l'âge saisi s'affiche dans la console de votre navigateur.

Utilisation de l'opérateur ternaire

Pour optimiser ce genre de logique décisionnelle, l'opérateur **ternaire** peut s'avérer pratique. Il permet d'écrire le code de manière plus concise. Voici comment réécrire l'exemple précédent avec cette approche :

```
    let age = 65;
    let message = (age < 3) ? 'Coucou mon bébé' :
                    (age < 18) ? 'Vous n\'avez pas le droit de
voter !' :
                (age < 30) ? 'Vous êtes encore jeune !' :
                (age < 62) ? 'Vous êtes encore actif !' :
                'Vous êtes certainement en retraite !';

    console.log(message);
```

Au début, il peut sembler complexe de comprendre ce qui se produit dans ce copde. Cependant, après un examen plus approfondi, il devient évident qu'il ne s'agit que d'une série de tests conditionnels ordinaires.
En conclusion, l'utilisation d'une séquence d'opérateurs tels que **if** et **else if** ou le choix de l'opérateur ternaire dépend du contexte et de la complexité de la logique mise en jeu. L'opérateur ternaire peut constituer un excellent choix pour des conditions simples, tandis que les instructions **if** et **else if** demeurent préférables pour des logiques plus élaborées.

L'Opérateur Double Point d'Interrogation (??)

L'opérateur double point d'interrogation (??), également appelé opérateur de **coalescence nulle**, a fait son apparition dans JavaScript avec la version ECMAScript 2020 (ES11). Cet opérateur permet de gérer les valeurs nulles ou indéfinies de façon concise, facilitant ainsi la gestion des valeurs par défaut dans le code.

Syntaxe
L'opérateur ?? s'utilise comme suit :

```
let valeur = variable ?? valeurParDefaut;
```

L'opérateur **??** permet d'évaluer deux expressions et de choisir celle à retourner en fonction de leur état. Il fonctionne ainsi : si **variable** n'est ni **nulle** ni **indéfinie**, alors la variable « **valeur** » recevra celle de **variable**. Dans le cas contraire, c'est-à-dire lorsque **variable** se trouve dans un de ces états, la variable **valeur** sera assignée à `valeurParDefaut`. Le résultat sera plus clair et plus protecteur pour le code, évitant ainsi les erreurs liées à l'utilisation de variables potentiellement vides.

Exemple 1 :

```
let variable = 5;
let valeurParDefaut = 10;
let valeur = variable ?? valeurParDefaut;

console.log(valeur); // Affiche 5
```

Ici, **variable** a la valeur 5, qui n'est ni **null** ni **undefined**. Donc, valeur est assignée à 5.

Exemple 2 : Valeur nulle

```
let variable = null;
let valeurParDefaut = 10;
let valeur = variable ?? valeurParDefaut;

console.log(valeur); // Affiche 10
```

Dans cet exemple, **variable** est **null**. Comme cette valeur est considérée comme "absence de valeur", valeur sera assignée à **valeurParDefaut**, qui est **10**.

Exemple 3 : valeur indéfinie

```
let variable;
let valeurParDefaut = 20;
```

```
let valeur = variable ?? valeurParDefaut;

console.log(valeur); // Affiche 20
```

Ici, **variable** est **undefined**. Par conséquent, valeur sera également assignée à **valeurParDefaut**, qui est **20**.

En résumé, l'opérateur double point d'interrogation ?? est un outil très utile qui répond à un vrai besoin de gestion des valeurs nulles ou indéfinies en JavaScript. Cette solution alternative est simple, claire et efficace dans la mesure où elle ne tient compte que des situations où une valeur est vraiment absente. Cela permet d'écrire du code plus propre et plus lisible tout en minimisant les risques d'erreurs que l'on peut rencontrer avec l'utilisation plus classique des opérateurs ||.

Conclusion

Les structures conditionnelles sont une des briques fondamentales de la programmation, déterminant le cheminement des instructions du programme en fonction de certaines conditions. Dans ce chapitre, nous avons examiné plusieurs façons différentes d'écrire ces structures en commençant par les instructions simples if, puis en passant à l'utilisation de else…if et else, et enfin à l'opérateur ternaire, qui permet de faire de même avec une syntaxe beaucoup plus concise. Enfin, nous avons également découvert l'opérateur double point d'interrogation ??, qui fournit une méthode simple pour tester les valeurs nulles et indéfinies. En plus des opérations logiques standards, cet opérateur fournit une approche plus claire et performante pour gérer les valeurs par défaut d'un programme.

Apprendre à utiliser ces structures conditionnelles est crucial pour les développeurs qui cherchent à écrire des programmes performants et responsifs. Celles-ci vous permettent de créer des applications capables de s'adapter à différentes situations et d'offrir une expérience interactive aux utilisateurs. En combinant les différents outils qui ont été abordés dans ce chapitre, vous pourrez écrire du code robuste et flexible pour une variété de scénarios d'utilisation.

À travers ce chapitre, nous vous avons présenté les principales méthodes de formulation des structures conditionnelles en JavaScript. Celles-ci vous permettent non seulement d'écrire des instructions de prise de décision logique, mais également d'améliorer la lisibilité du code et de rendre vos applications plus conviviales. Avec ces connaissances et la solide base qu'elle vous offre, vous serez plus en

mesure de comprendre des concepts plus avancés abordés dans les prochains chapitres.

Chapitre 5

Les boucles en JavaScript

En programmation, il est souvent nécessaire de répéter une action de manière automatique. Ainsi, sans boucles, pour répéter la tâche, il faudrait copier et coller le code pour chaque répétition, mais ce n'est ni efficace ni pratique. Les boucles sont des structures de contrôle qui permettent d'automatiser l'exécution d'un bloc de code en le répétant plusieurs fois aussi longtemps qu'une condition donnée est vraie.
En JavaScript, il existe plusieurs types de boucles qui vous permettront de gérer des tâches répétitives. En utilisant les boucles, vous pouvez automatiser des processus sans avoir à écrire des dizaines et des centaines de lignes. En outre, ils sont tout aussi importants pour le traitement des données, la gestion des interactions ou le traitement des variables. Dans ce chapitre, nous définissons tous les types de boucles, y compris :

- La boucle **for**, qui doit être utilisée lorsqu'un grand nombre de répétitions doit être effectué;
- La boucle **while**, exécutée tant que la condition donnée est vraie;
- La boucle **do… while** qui exécute au moins une fois avant de vérifier la condition.

En connaissant ces types, vous pourrez écrire un code plus dynamique et plus performant.

La boucle for

La boucle for est idéale lorsque vous savez à l'avance combien de fois vous devez répéter un bloc de code. C'est une boucle déterminée : vous pouvez spécifier le nombre d'itérations avant même que la boucle ne démarre.

Syntaxe :

```
for (initialisation; condition; incrémentation) {
    // Code à exécuter à chaque itération
}
```

Initialisation :

- Cette étape se produit une seule fois au début de la boucle. Elle permet de définir et d'initialiser une variable de contrôle (souvent une variable compteur) qui sera utilisée pour suivre le nombre d'itérations.
- Exemple : let i = 0 crée la variable i et la définit à 0 avant que la boucle ne commence.

Condition :

- La condition est évaluée avant chaque itération de la boucle. Aussi longtemps que cette condition est vraie (true), le bloc de code à l'intérieur de la boucle sera exécuté.
- Si la condition devient fausse (false), la boucle s'arrête et le programme continue après la boucle.
- Par exemple : i < 5 vérifie si i est inférieur à 5. Si c'est vrai, la boucle continue, sinon elle s'arrête.

Incrémentation :

- L'incrémentation a lieu à la fin de chaque itération. Elle permet de modifier la variable de contrôle pour se rapprocher de la condition d'arrêt de la boucle.
- Par exemple : i++ augmente la valeur de i de 1 après chaque itération.

Exemple :

```
for (let i = 0; i < 5; i++) {
    console.log("L'itération numéro : " + i);
}
```

Explication :

- Initialisation (let i = 0) : La variable i est définie à zéro, ce qui signifie que la boucle débute avec i égal à 0.
- Condition (i < 5) : tan que i demeure inférieur à 5, la boucle continue à s'exécuter de façon répétée.
- Incrémentation (i++) : À chaque tour, on ajoute 1 à la valeur de i.
- À chaque passage dans la boucle, i prend une valeur différente (de 0 à 4) et le message s'affiche avec cette valeur changée.

Le résultat sera :

```
L'itération numéro : 0
L'itération numéro : 1
L'itération numéro : 2
L'itération numéro : 3
L'itération numéro : 4
```

La boucle while

La boucle while est une structure de contrôle qui permet de répéter l'exécution d'un bloc de code aussi longtemps qu'une condition donnée est vérifiée. Contrairement à la boucle for, dont le nombre d'itérations est déterminé à l'avance, la boucle while peut s'exécuter un nombre indéfini de fois en fonction de l'évaluation de la condition à chaque tour.

Syntaxe :

```
while (condition) {
    Bloc de code à exécuter tant que la condition est vraie
}
```

Explication :

- La condition est vérifiée au début de chaque itération. Si elle est vraie (valeur de vrai), le bloc de code à l'intérieur de la boucle est exécuté.

- À l'inverse, si la condition est fausse (faux), la boucle

prend fin et le programme continue après celle-ci.

- La condition doit être modifiée au sein même de la boucle (par exemple en incrémentant une variable), autrement la boucle risque de ne jamais s'arrêter et de devenir infinie.

Exemple :

```
let compteur = 0;

while (compteur < 3) {
    console.log("Compteur : " + compteur);
     compteur++; // On augmente la valeur de compteur à chaque
itération
}
```

Explication de l'exemple :

- Condition : La boucle débute avec le compteur établi à 0 et la vérification de la condition si le compteur est inférieur à 3.
- Si la valeur du compteur est inférieure à 3, le bloc de code à l'intérieur de la boucle continue à être exécuté à chaque itération. Dans le cas contraire, la boucle prend fin.
- À chaque répétition, le message "Compteur : " suivi de la valeur attribuée au compteur s'affiche puis la variable **compteur** augmente d'une unité grâce à l'instruction **compteur++.**
- Lorsque le compteur atteint 3, la condition compteur < 3 devient fausse, mettant fin au déroulement de la boucle.

Résultat dans la console :

```
Compteur : 0
Compteur : 1
Compteur : 2
```

Attention aux boucles infinies :

Il est impératif de modifier la condition à l'intérieur de la boucle, car sinon vous pourriez créer une boucle sans fin. Par exemple, dans le code mentionné plus haut, si vous omettez d'incrémenter le compteur (en supprimant compteur++), la condition compteur < 3 restera toujours vraie, et la boucle ne s'arrêtera jamais.

Utilisations typiques de la boucle while :

- La boucle **while** est idéale lorsqu'il est impossible de prédire à l'avance le nombre exact de répétitions du bloc d'instructions.
- Elle est souvent employée lorsqu'il faut attendre qu'une condition particulière soit remplie avant d'interrompre l'exécution.
- Par exemple, on pourrait patienter qu'une donnée spécifique soit reçue ou qu'une action déterminée soit menée à bien avant d'abandonner la boucle.

La boucle do...while

La boucle **do...while** garantit qu'un bloc de code s'exécute au moins une fois, avant d'évaluer si la condition est remplie ou non. Contrairement à la boucle **while** classique, dont la condition peut ne jamais être vérifiée si elle est fausse dès le départ, la structure **do...while** assure une première exécution du bloc. Par la suite, la condition est testée après chaque itération du bloc : si elle est vraie, le code à l'intérieur de la boucle s'exécute à nouveau, dans le cas contraire le programme saute hors de la boucle.

Syntaxe :

```
do {
    Bloc de code à exécuter
} while (condition)
```

Explication :

- Le code placé dans la boucle (entre les accolades) s'exécute immédiatement, sans vérifier la condition.

- À la fin de chaque répétition, la condition logique est évaluée. Si elle est vraie, une nouvelle itération a lieu; dans le cas contraire, le programme quitte la boucle do...while.

Exemple concret :

```
let compteur = 5;

do {
    console.log("Compteur : " + compteur);
    compteur++;
} while (compteur < 3);
```

Explication de l'exemple :

Étape 1 : Exécution du bloc : Bien que la condition sur le compteur ne soit déjà pas remplie (sa valeur étant de 5), la boucle do...while débute tout de même par exécuter une fois le code présent dans son bloc.

Étape 2 : Vérification de la condition : Après cette première exécution, la condition est examinée. Ici, le compteur valant désormais 6 du fait de l'incrémentation, la condition compteur < 3 s'avère fausse, ce qui met un terme à la boucle.

Résultat dans la console :

```
Compteur : 5
```

Remarque : Même si la condition était incorrecte dès le départ, le message "Compteur : 5" est quand même affiché, car dans une boucle do...while le bloc est de toute façon toujours exécuté une fois, avant toute vérification de la condition.

Ainsi, dans cet exemple précis, la boucle do...while assure que le code sera au minimum exécuté une fois, et ce peu importe que la condition compteur < 5 soit initialement erronée. Une boucle while classique n'aurait quant à elle pas réalisé cette opération du fait du caractère faux de la condition dès le départ.

Modifions maintenant la condition dans l'exemple précèdent :

```
let compteur = 5;

do {
```

```
    console.log("Compteur : " + compteur);
    compteur++;
} while (compteur < 10);
```

- La variable compteur est initialisée à 5.
- La boucle **do...while** exécute d'abord le bloc de code entre les accolades avant de vérifier la condition. Ici, elle affiche la valeur actuelle de la variable « compteur » dans la console (5), puis incrémente la variable compte de 1.
- À chaque itération, la condition compteur < 10 est vérifiée. Si elle est vraie, le bloc de code est exécutée de nouveau. Sinon, la boucle s'arrête.
- À chaque tour, la variable compteur augmente de 1. Ainsi, après la première répétition, compteur passe à 6 puis 7 et ainsi de suite jusqu'à 10.
- La boucle continue tant que la condition compteur < 10 est vérifiée. Lorsque compteur atteint 10, la condition devient fausse (10 n'est pas inférieur à 10) et la boucle prend fin.

Résultat attendu dans la console :

```
Compteur : 5
Compteur : 6
Compteur : 7
Compteur : 8
Compteur : 9
```

Contrôle de boucle : break et continue

En JavaScript, vous pouvez gérer le déroulement des boucles à l'aide des mots-clés **break** et **continue**.

break : Ce mot-clé permet d'interrompre prématurément l'exécution de la boucle avant qu'elle ne se termine naturellement. Lorsqu'il est rencontré, la boucle s'arrête immédiatement.

Exemple :

```
for (let i = 0; i < 5; i++) {
    if (i === 3) {
        break; // Arrête la boucle quand i vaut 3
    }
    console.log("La valeur de i est: " + i);
}
```

Explication : La boucle se déroule normalement jusqu'à ce que la variable **i** atteigne la valeur 3. À ce moment, **break** met un terme à la boucle, de sorte que le programme s'achève avant d'afficher les valeurs 3 et 4.

Résultat dans la console :

```
La valeur de i est: 0
La valeur de i est: 1
La valeur de i est: 2
```

continue : Ce mot-clé permet de passer directement à l'itération suivante, sans exécuter le reste du code de la boucle pour l'itération en cours.

Exemple :

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        continue; // Ignore le reste du code pour i = 2
    }
    console.log("La valeur de i est: " + i);
}
```

Explication : Lorsque la valeur de i est 2, le mot-clé **continue** est activé, ce qui force la boucle à sauter l'affichage de cette valeur. Les autres valeurs sont affichées normalement.

Le résultat sera :

```
La valeur de i est: 0
La valeur de i est: 1
La valeur de i est: 3
La valeur de i est: 4
```

Résumé des boucles en JavaScript

En JavaScript, les boucles permettent de répéter l'exécution d'un bloc de code en fonction d'une condition. Il existe trois types principaux de boucles, chacune adaptée à des situations spécifiques :

La boucle **for** est utilisée lorsque le nombre d'itérations est connu à l'avance. Elle convient parfaitement aux situations où vous devez répéter une action un certain nombre de fois, avec une gestion explicite de l'initialisation, de la condition et de l'incrémentation.

La boucle **while** est employée lorsque vous ne savez pas exactement combien de fois la boucle devra s'exécuter. Elle continue tant qu'une condition demeure vraie, et elle est vérifiée avant chaque itération. Elle convient aux situations où la condition peut varier pendant l'exécution.

La boucle **do...while** fonctionne de manière similaire à la boucle **while**, mais avec une différence importante : elle garantit que le bloc de code sera exécuté au moins une fois, avant que la condition ne soit testée. Elle est utile lorsque vous souhaitez exécuter un bloc de code avant de vérifier la condition.

Conclusion

Bien que les boucles soient essentielles pour automatiser les tâches répétitives, il est important de les utiliser avec parcimonie afin de ne pas alourdir inutilement le code. Dans ce chapitre, nous nous sommes concentrés sur les boucles les plus élémentaires de JavaScript - la boucle for, la boucle while et la boucle do...while. Chacune répond à des besoins précis selon la manière dont vous souhaitez gérer les itérations. Que ce soit pour des répétitions définies à l'avance grâce à for, des boucles indéterminés avec while ou un traitement systématique garanti avec do...while, ces structures vous permettent de simplifier le déroulement de tâches récurrentes.

Il existe cependant d'autres patterns itératifs comme les boucles

for...in et **for...of** qui autorisent les parcours **d'objets** et de **collections**. Nous détaillerons ces techniques plus évoluées dans les chapitres à venir, lorsque nous aborderons les notions fondamentales de tableaux et d'objets JavaScript.

A présent que vous maitrisez les rouages de base des boucles, vous pouvez produire un code plus performant, adaptable et dynamique. Cependant gardez à l'esprit que la lisibilité et la maintenabilité sont aussi essentielles qu'une exécution optimale.

Chapitre 6

Les fonctions en JavaScript

Les fonctions représentent l'un des concepts fondamentaux et puissants de JavaScript. Elles permettent de regrouper le code dans un bloc que l'on peut exécuter à plusieurs reprises sans devoir le réécrire à chaque fois. Les fonctions favorisent la réutilisation du code et aident à structurer les programmes de manière plus modulaire et organisée.

Au sein d'une fonction, il est possible de définir des paramètres, de transmettre des valeurs spécifiques lors de l'exécution, et de retourner des résultats. Cela rend les programmes plus souples et dynamiques. En JavaScript, les fonctions peuvent être simples ou extrêmement complexes, et elles sont essentielles pour des tâches telles que la gestion des événements, les calculs et l'automatisation de processus répétitifs.

Ce chapitre vous guidera à travers les bases des fonctions en JavaScript, de leur déclaration jusqu'à leur utilisation, en passant par les paramètres et les valeurs de retour. Vous découvrirez également les fonctions anonymes et les fonctions fléchées, des formes avancées permettant d'écrire le code de manière plus concise et lisible.

Êtes-vous prêt à apprendre comment organiser et réutiliser votre code efficacement grâce aux fonctions?

Quelques fonctions natives en JavaScript : alert(), prompt(), et confirm()

Avant de vous lancer dans la création de vos propres fonctions, il vaut mieux se familiariser avec certaines fonctions natives déjà incluses en JavaScript. Ces fonctions vous permettent d'interagir facilement avec l'utilisateur et sont très utiles pour concevoir des interactions de base.

Dans les chapitres précédents, nous avons déjà utilisé la fonction d'alerte. Regardons comment elle fonctionne ainsi que deux autres fonctions essentielles : question et confirmation.

La fonction alert() :

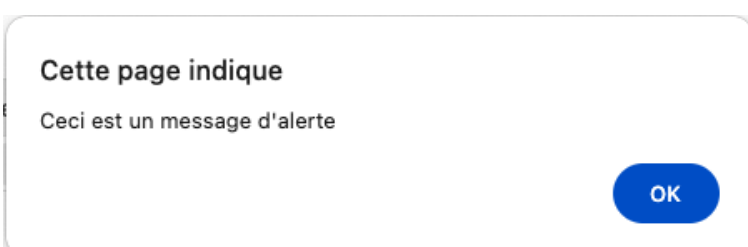
La fonction d'alerte sert à afficher un message simple à l'utilisateur dans une boîte de dialogue modale. Elle est employée pour informer l'utilisateur d'un événement ou d'une situation, mais n'attend pas de réponse de sa part. L'utilisateur doit cliquer sur le bouton "OK" pour fermer la boîte de dialogue et poursuivre.

Elle est souvent utilisée pour des notifications ou pour des tests rapides de débogage.

Nous avons déjà eu l'occasion de l'employer à plusieurs reprises. Pour vous remettre les idées en place, voici un rappel de sa syntaxe via l'exemple suivant :

```
    alert("Ceci est un message d'alerte");
```

L'exécution de ce code affichera la boite d'alerte suivante :



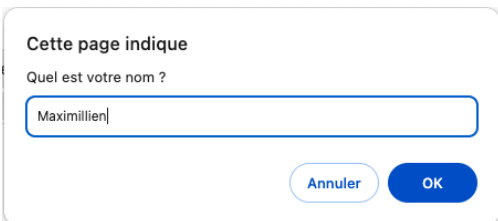
La fonction prompt()
La fonction prompt() offre une interaction plus directe avec l'utilisateur. Elle affiche une boîte de dialogue demandant à l'utilisateur de saisir des informations (telles qu'un texte ou un

nombre). Cette fonction renvoie la valeur saisie par l'utilisateur sous forme de chaîne de caractères, ou null si l'utilisateur annule la demande.

Syntaxe :

```
let myName = prompt("Quel est votre nom ?");
```

Dans cet exemple, une boîte de dialogue s'ouvre invitant l'utilisateur à entrer son nom. Si vous exécutez cette ligne de code, une boîte de dialogue apparaîtra. Si l'utilisateur saisit un nom et clique sur le bouton « OK », la variable myName contiendra la valeur saisie. Si l'utilisateur annule, la variable myName sera nulle



Prompt() est utile pour recueillir de simples données auprès de l'utilisateur, comme des noms, des âges ou des préférences.

La fonction confirm()

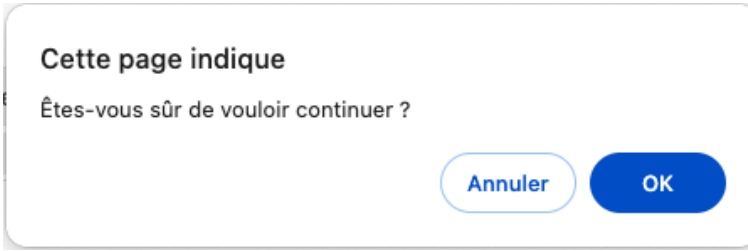
La fonction **confirm()** affiche une boîte de dialogue contenant un message et deux boutons: Valider et Annuler. Elle renvoie une valeur booléenne : vrai (true) si l'utilisateur valide, et faux (false) s'il annule. Cette fonction convient pour obtenir une confirmation avant d'effectuer une action.

Syntaxe :

```
let response = confirm("Êtes-vous sûr de vouloir continuer ?");
```

Cet code montre comment afficher une boîte de dialogue demandant à l'utilisateur de confirmer s'il souhaite continuer. Si l'utilisateur clique sur "OK", la variable response contiendra (true) vrai. Si l'utilisateur clique sur "Annuler", la variable contiendra (false) faux. La fonction **confirm()** est fréquemment utilisée pour demander une confirmation

avant l'exécution d'une action importante, comme supprimer un fichier ou quitter une page.



Ces fonctions représentent des outils simples mais puissants pour créer des interactions basiques avec l'utilisateur en JavaScript. Elles peuvent sembler élémentaires mais s'avèrent extrêmement utiles pour saisir la manière dont JavaScript peut communiquer avec l'utilisateur. À présent que vous maîtrisez ces commandes, nous allons passer à la création de vos propres fonctions pour organiser et réutiliser votre code.

Déclaration et création de fonctions en JavaScript

Après avoir pris connaissance des fonctionnalités intégrées comme alert(), prompt() et confirm(), il est temps de découvrir comment créer vos propres fonctions en JavaScript. Une fonction personnalisée vous permet de rassembler une série d'instructions sous un nom unique et de les exécuter autant de fois que nécessaire. Cela vous aide à structurer et à réutiliser votre code de manière efficace.

Syntaxe de la déclaration d'une fonction

Une fonction JavaScript se définit à l'aide du mot-clé **function**, suivi du nom de la fonction, de parenthèses (où vous pouvez spécifier des paramètres), puis d'accolades contenant le bloc de code à exécuter.
Voici la syntaxe de base pour déclarer une fonction :

```
function nomDeLaFonction( ) {
    // Code à exécuter
}
```

Une fois la fonction définie, vous pouvez l'appeler (ou "exécuter" la fonction) en écrivant son nom suivi de parenthèses.

```
nomDeLaFonction( );
```

Exemple simple
Créons une fonction qui affiche un message de bienvenue :

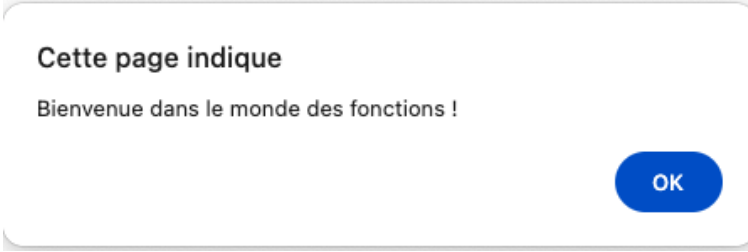
```
function displayMessage() {
    alert("Bienvenue dans le monde des fonctions !");
}
```

Explication :

Le programme commence à définir une fonction nommée **displayMessage** sans paramètres. Cette fonction contient une ligne de code unique appelant la méthode **alert** pour afficher un message à l'utilisateur.

```
displayMessage(); // Cela déclenche l'affichage du message.
```

Lorsque vous exécutez ce code, vous obtiendrez l'affichage de la boite d'alerte suivante :



Fonctions avec paramètres

Les fonctions peuvent également accepter des paramètres (à mettre entre parenthèses), que l'on peut transmettre lors de l'appel pour personnaliser leur comportement.

Exemple :

```
function greet(name) {
    alert("Bonjour, " + name + " !");
}
```

Explication :

La fonction **greet** définie ici prend un seul paramètre nommé **name**. Lorsqu'on l'appelle en transmettant une valeur comme "Alex", le message d'alerte affiche "Bonjour, Alex!".

```
greet("Alex");  // Affiche : "Bonjour, Alex !"
greet("Marie"); // Affiche : "Bonjour, Marie !"
```

Deux appels de la focntion **greet** : le premier appel avec "Alex" comme valeur du paramètre name, le second avec "Marie", générant des messages personnalisés grâce au paramètre.

Cette flexibilité offerte par les paramètres permet d'adapter le fonctionnement des fonctions aux valeurs transmises, les rendant beaucoup plus dynamiques et réutilisables.

Retourner une valeur depuis une fonction

Les fonctions peuvent également retourner une valeur au moyen du mot-clé retour. Cela se révèle très utile pour récupérer un résultat calculé par la fonction et l'utiliser plus tard dans le programme.

Exemple :

```
function addition(a, b) {
    return a + b;
}
```

Cette fonction prend deux paramètres, a et b, et renvoie la somme de ces deux nombres au moyen du mot clé **return**.

Le mot-clé **return** met fin immédiatement à l'exécution de la fonction et renvoie la valeur spécifiée.

Vous pouvez appeler la fonction et stocker sa sortie dans une variable :

```
function addition(a, b) {
    return a + b;
}
```

```
let result = addition(5, 3);
console.log(result); // Affiche : 8
```

Lors de l'exécution de ce code, la valeur renvoyée par la fonction **addition** est affectée à la variable **result**, puis affichée dans la console.

Fonctions avec plusieurs paramètres

Une fonction peut accepter plusieurs paramètres, ce qui vous permet de gérer plusieurs données en même temps.

Exemple :

```
function multiply(a, b) {
    return a * b;
}

let result = multiply(4, 7);
console.log(result); // Affiche : 28
```

Ici, nous avons aussi une fonction nommée multiply() qui prend deux paramètres a et b (elle peut en avoir plusieurs), et renvoie le produit de ces deux valeurs.

Fonctions avec Paramètres par Défaut en JavaScript

En JavaScript, il est possible de définir des paramètres par défaut pour les fonctions. Cela permet de spécifier une valeur qui sera utilisée si aucun argument n'est fourni lors de l'appel de la fonction. Cette capacité facilite la gestion des valeurs manquantes et rend le code plus clair.

Syntaxe des Paramètres par Défaut
La syntaxe pour définir un paramètre par défaut consiste à attribuer une valeur à un paramètre dans la déclaration de la fonction. Voici un exemple :

```
function greet(name = "Invité") {
```

```
        console.log("Bonjour, " + name + " !");
    }
```

Dans cet exemple, le paramètre **name** a une valeur prédéterminée de "Invité". Cela signifie que si la fonction est appelée sans argument, elle affichera "Bonjour, Invité!".

Appel sans argument :

```
function greet(name = "Invité") {
        console.log("Bonjour, " + name + " !");
    }
    greet();  // Affiche : "Bonjour, Invité !"
```

Lorsque vous exécutez ce code, vous obtiendrez dans la console de votre navigateur l'affichage : « Bonjour Invité ! ».

Appel avec un argument :

```
    function greet(name = "Invité") {
        console.log("Bonjour, " + name + " !");
    }
    greet("Alex");  // Affiche : "Bonjour, Alex !"
```

Dans ce cas, si vous exécutez ce code, vous obtiendrez dans la console de votre navigateur l'affichage : « Bonjour Alex! ».

Avantages des paramètres par défaut

- **Simplicité** : Permet d'éviter les vérifications manuelles pour déterminer si un argument a été fourni.
- **Flexibilité** : Facilite l'appel de la fonction avec différents nombres d'arguments.
- **Clarté** : Indique clairement quelles valeurs sont prédéterminées.

Combinaison avec d'autres Paramètres

Bien que les paramètres par défaut puissent être combinés avec des paramètres normaux, ils présentent certains avantages. Prenons l'exemple suivant :

```
    function greet(name, greeting = "Bonjour") {
        console.log(greeting + ", " + name + " !");
    }
```

```
greet("Alex");           // Affiche : "Bonjour, Alex !"
greet("Marie", "Salut"); // Affiche : "Salut, Marie !"
```

Dans cet exemple, le paramètre « **greeting** » dispose d'une valeur par défaut de «Bonjour», mais celui-ci peut être remplacé si nécessaire par un autre message.

Résultat :

```
Bonjour, Alex !
Salut, Marie !
```

En résumé, les paramètres par défaut en JavaScript constituent une fonctionnalité puissante qui simplifie la gestion des fonctions tout en améliorant la lisibilité du code. Ils permettent aux développeurs de concevoir des fonctions plus flexibles et de mieux gérer les cas où les valeurs d'arguments ne seraient pas fournies.

Importance des fonctions en JavaScript

Les fonctions jouent un rôle central dans l'écriture de programmes efficaces et bien structurés. Elles permettent de :

- **Réutiliser du code** : Une fois que vous avez créé une fonction, vous pouvez l'utiliser autant de fois que nécessaire, sans avoir à répéter les mêmes instructions.
- **Rendre le code plus clair** : En isolant certaines parties de votre programme dans des fonctions bien nommées, vous améliorez la lisibilité et la compréhension de votre code.
- **Modularité** : Les fonctions permettent de découper des tâches complexes en plusieurs petites unités, facilitant ainsi le développement, la maintenance et le débogage.
- **Flexibilité** : Avec les paramètres, les fonctions deviennent dynamiques et peuvent s'adapter aux données qu'elles reçoivent.

Portée des variables et fonctions en JavaScript

La portée des variables JavaScript détermine où une certaine variable

peut être utilisée dans le code. Il est essentiel de bien comprendre la portée des variables pour éviter les conflits de noms, puis gérer la visibilité des variables.

Portée globale

Toute variable déclarée en dehors d'une fonction est une variable globale. En d'autres mots, cela signifie qu'elles sont accessibles partout.

Exemple :

```
var globalVar = "Je suis global";

function showGlobal() {
    console.log(globalVar); // Affiche : "Je suis global"
}

showGlobal();
```

Dans cet exemple, la variable « globalVar » est déclarée en dehors de la fonction showGlobal(). En d'autres termes, elle est déclarée globale. Lors de l'exécution de ce code, dans la console de votre navigateur, vous aurez le rendu suivant : « Je suis global ». Cela signifie que la variable « globalVar « a bel été utilisée dans la fonction showGlobal() sans aucun souci.

```
Je suis global
```

Portée locale

Une variable déclarée à l'intérieur d'une fonction a une portée locale. Cela signifie qu'elle ne peut être utilisée dans une autre fonction et est supprimée une fois la fonction terminée.

Exemple

```
function showLocal() {
    var localVar = "Je suis local";
    console.log(localVar); // Affiche : "Je suis local"
}
```

```
    showLocal();
     console.log(localVar); // Provoque une erreur : localVar
n'est pas défini
```

Dans la focntion ShowLocal(), la variable “localVar” est déclarée en utilisant “var”, elle est donc uniquement accessible localement à la fonction elle-même. Par conséquent, quand ShowLocal() est exécuté, “Je suis local” apparaît sur le console.

```
Je suis local
```

Cependant, en dehors de la fonction, la dernière ligne de code tente d’afficher localVar dans la console, sans succès. Puisque sa portée est locale, elle n'est pas définie en dehors dela fonction showLocal().

```
▶ Uncaught ReferenceError: localVar is not defined
    at my_script.js:7:13
```

Ce code illustre donc bien comment les variables locales en JavaScript ne peuvent être utilisées qu'à l'intérieur de leur fonction de déclaration.

Fonctions récursives en JavaScript

Une fonction récursive a la particularité de s’appeler elle-même dans sa définition. Les fonctions récursives sont particulièrement pratiques pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires, comme ceux qui impliquent des calculs mathématiques ou le traitement des structures de données arborescentes, entre autres.

Principe de la récursion
Une fonction récursive doit généralement avoir deux éléments clés :

- **Cas de base** : On doit avoir une condition dont le rôle est d'arrêter la récursion à un certain moment pour éviter une boucle infinie.
- **Appel récursif** : La fonction s'appelle elle-même avec des arguments modifiés.

Afin d’éclaircir cette notion de récursivité, prenons l’exemple de calcul de la somme des n premiers entiers : 1 + 2 + 3 + 4 + … + n, où n

est un entier positif. Voici une fonction qui calcule cette somme en javascript.

Voici un exemple de fonction qui réalise ce calcul :

```
function sumCalculation(n) {
    let sum = 0;
    for (i = 0; i <= n; i++) {
        sum += i;
    }
    return sum;
}

console.log(sumCalculation(5));//donne 15
```

L'exécution de ce code affiche bien, dans la console de votre navigateur, la somme des entiers de 0 à 5.

Mais, nous pouvons également résoudre de genre de d'exercices en créant une fonction **récursive**.

```
function sumRecursif(n) {
    if (n === 0) {
        return 0;
    } else {
        return n + sumRecursif(n - 1);
    }
}

// Exemple d'utilisation :
console.log(sumRecursif(5)); // Résultat : 15
```

Ci-dessous, un exemple de code qui montre comment une fonction, sumRecursif, calcule la somme des n premiers entiers, en utilisant une fonction récursive. Lorsque n devient égal à zéro, la fonction renvoie 0 (zéro), sinon elle retourne n ajouté à la somme des entiers de 0 à n-1 obtenue en rappelant la fonction sur n-1.
À titre d'exemple, si l'on appelle la fonction sumRecursif(5), nous aurons :

- sumRecursif(5) appelle 5 + sumRecursif(4)

- sumRecursif(4) appelle 4 + sumRecursif(3)
- sumRecursif(3) appelle 3 + sumRecursif(2)
- sumRecursif(2) appelle 2 + sumRecursif(1)
- sumRecursif(1) appelle 1 + sumRecursif(0)
- sumRecursif(0) renvoie 0

En remontant dans les appels récursifs, nous avons :

- sumRecursif(1) renvoie 1 + 0 = 1
- sumRecursif(2) renvoie 2 + 1 = 3
- sumRecursif(3) renvoie 3 + 3 = 6
- sumRecursif(4) renvoie 4 + 6 = 10
- sumRecursif(5) renvoie 5 + 10 = 1

Toutefois, il est à noter qu'en général, les fonctions récursives ont tendance à consommer beaucoup plus de ressources que celles utilisant des boucles. Dès lors, il est à garder à l'esprit de les utiliser avec le minimum de récursivité possible ou de les optimiser.

Conclusion

Ce chapitre nous a permis de découvrir les détails des fonctions en JavaScript. En fait, les fonctions, comme nous l'avions précisé, jouent un rôle important en programmation en général et en JavaScript en particulier. C'est grâce aux fonctions qu'on peut structurer et organiser le code de manière efficace. Dans ce chapitre, nous avons commencé par expliquer quelques fonctions natives en JavaScript telles que les fonctions alert, prompt et confirm qui facilitent le dialogue entre votre script et l'utilisateur.

Ensuite, nous avons montré comment déclarer et appeler des fonctions, en soulignant l'importance des paramètres, notamment les paramètres par défaut qui procurent quelque souplesse supplémentaire pour gérer les arguments.

Puis nous avons introduit les fonctions récursives, qui sont particulièrement utiles pour résoudre des problèmes complexes en les décomposant en sous problèmes plus simples. Cependant, nous avons souligné également qu'il faut toujours recourir à la récursivité avec prudence car ça peut alourdir les performances.

Pour résumer, les fonctions sont un outil puissant en JavaScript pour écrire du code clair, modulable et réutilisable. C'est nécessaire pour tout développeur qui veut réaliser des applications web robustes et performantes.

Chapitre 7

Les tableaux en JavaScript

Les tableaux sont des structures de données indispensable en JavaScript, car il vous permet de mémoriser plusieurs valeurs dans une seule variable et de les manipuler. Comparés aux variables simples, qui ne peuvent contenir qu'une seule valeur à la fois, un tableau peut contenir une collection d'éléments comme des nombres, des chaînes de caractères ou même d'autres tableaux.
Des tableaux s'imposent dès que vous avez à manipuler des données dynamiques à plusieurs niveaux, que ce soit des listes d'objets, des résultats de calculs ou toute autre chose. Munis des méthodes propres et des propriétés qui leur sont associées, ils vous permettent de traiter ces ensembles de données de manière efficace et sont un élément central dans le développement d'applications.
À l'issue de ce chapitre, vous saurez :

- Comment créer un tableau,
- Comment y accéder et en modifier les éléments,
- Comment le parcourir de bout en bout avec des boucles,
- Et comment manipuler ses données en utilisant des méthodes et des propriétés propres aux tableaux.

Maîtriser les tableaux vous permet de gérer des ensembles de données de façon souple et tout en laissant votre code plus concis et modulaire. Êtes-vous prêt à découvrir comment manipuler les tableaux en JavaScript ?

Création des tableaux en JavaScript

En JS, un tableau (ou array en anglais) est une structure de donnée qui peut contenir plusieurs valeurs dans une seule variable. Ces valeurs peuvent être de tous les types, tels que des nombres, des chaînes de caractères, des booléens ou même d'autres tableaux.

Il y a plusieurs façons de créer un tableau en JavaScript. Les principales sont: l'emploi de littéraux de tableaux, et l'utilisation du constructeur Array().

Utilisation des littéraux de tableaux

La façon la plus simple et habituelle de créer un tableau consiste à utiliser les symboles de crochets. Les éléments du tableau sont simplement mis entre crochets [ ] et séparés par des virgules.

Syntaxe :

```
let myArray = [value1, value2, value3, ...];
```

Explication détaillée :

- myArray est le nom de la variable qui va contenir le tableau. Choisissez n'importe quel nom significatif pour bien représenter votre tableau. Le mot clé **let** vous permet de déclarer cette variable flexible (elle peut être réassignée si nécessaire).
- [value1, value2, value3,...] : les crochets [ ] indiquent que vous créez un tableau.

À l' intérieur des crochets vous énumérer les éléments du tableau. Toutes ces valeurs (ici représentées par value1, value2, etc.) peuvent être de toutes sortes de chose. Elles peuvent être des chaînes de caractères, des nombres, ou un autre tableau par exemple.

Exemple :

```
let fruits = ["Apple", "Banana", "Orange"];
```

Explication :

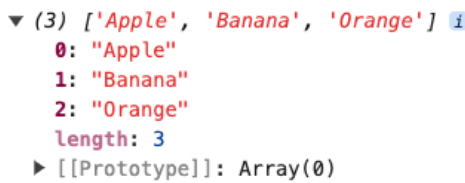
- Dans cet exemple, nous avons créé un tableau nommé fruits contenant trois éléments : "Apple", "Banana", et "Orange". Chaque élément est séparé par une virgule.

- Pour accéder aux éléments d'un tableau, il suffit d'utiliser leur indice (leur position), qui commence toujours à 0. Par exemple, fruits[0] pointe sur l'élément "Apple", qui est le premier élément du tableau.

Dans vous modifiez l'exemple précèdent comme suit :

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits);
```

Lorsque vous exécutez ce code, vous obtiendrez, dans la console de votre navigateur, l'affichage suivant :



Il est à noter que le tableau "fruits" est ainsi affiché en entier avec ses trois éléments encadrés par des crochets [ ] qui indique qu'il s'agit bien d'un tableau.

Ce résultat affiche également chaque élément du tableau, en indiquant son numéro (l'indice), qui commence toujours à 0. C'est pourquoi l'élément "Apple" a pour indice 0, "Banana" a pour indice 1, et ainsi de suite. Sa longueur totale est enfin affichée, dans ce cas, le tableau a trois éléments.

On peut aller plus loin et présenter le contenu du tableau de façon plus structurée en guise de tableau dans la console. Pour ce faire, modifions le code précédent comme suit :

```
let fruits = ["Apple", "Banana", "Orange"];
console.table(fruits);
```

Lorsque vous exécutez ce code, vous obtiendrez :

Observez attentivement les indices associés à chaque élément du tableau. Ces indices débutent à partir de 0 et c'est précisément grâce à

ces indices que nous pourrons accéder aux différents éléments du tableau.

Utilisation de la fonction Array()

Un autre moyen de créer un tableau consiste à utiliser le constructeur Array(),bien que cette approche soit moins couramment utilisée.

Syntaxe :

```
let myArray = new Array(value1, value2, value3, ...);
```

let myArray :

- **myArray** est le nom de la variable qui contiendra le tableau. Vous pourrez donner à cette variable n'importe quel un nom capable de rendre compte de ce que représente le tableau. L'utilisation de **let** permet de déclarer cette variable de façon flexible (elle peut être réassignée si nécessaire).

new Array() :

- Ici, Array() est un constructeur intégré en JavaScript qui crée un tableau. Le mot-clé **new** est utilisé pour indiquer que vous créez une nouvelle instance d'un tableau.

(value1, value2, value3, ...) :

- Les paramètres que vous passez dans le constructeur Array() sont les éléments du tableau. Ils peuvent être de tout type (nombres, chaînes de caractères, booléens, objets, etc.).

Exemple :

```
let numbers = new Array(10, 20, 30);
```

Explication :
Le tableau numbers est créé en utilisant le constructeur Array() avec les valeurs 10, 20 et 30.

Remarque : cette méthode peut être particulièrement utile si l'on veut initialiser un tableau ayant une certaine longueur, mais sans lister tous

ses éléments immédiatement.

Tableaux vides

Il est également possible de créer un tableau vide, que l'on remplit plus tard au gré des besoins du programme.

Exemple :

```
let shoppingList = [];
```

Ici, **shoppingList** est un tableau vide. Comme nous allons le voir, vous pouvez ajouter des éléments plus tard en utilisant des méthodes comme **push()** ou en attribuant des valeurs à des indices spécifiques.

Les tableaux en JavaScript sont faciles à créer et à utiliser. On peut les initialiser avec des valeurs précises ou créer des tableaux vides à remplir plus tard. Les littéraux de tableau (Utilisation des crochets) sont la méthode la plus courante, tandis que l'utilisation du constructeur Array() est utile dans des situations particulières.

Dans la section suivante, nous verrons comment accéder aux éléments d'un tableau, les modifier, et utiliser les méthodes appropriées aux tableaux.

Comment accéder aux éléments d'un tableau en JavaScript ?

Lorsqu'on travaille avec des tableaux en JavaScript, il est essentiel de savoir comment accéder aux éléments qu'ils contiennent. Chaque élément dans un tableau a une position appelée indice (ou index en anglais), et c'est grâce à cet indice qu'on peut accéder à un élément précis.

Accès par indice
Les indices dans un tableau en JavaScript commencent toujours à 0. Cela signifie que le premier élément du tableau a l'indice 0, le deuxième élément a l'indice 1, et ainsi de suite.
Exemple :

```
let myArray = ["Apple", "Banana", "Orange"];
console.log(myArray[0]);  // Affiche "Apple"
console.log(myArray[1]);  // Affiche "Banana"
```

```
    console.log(myArray[2]);  // Affiche "Orange"
```

Dans cet exemple :

- L'instruction myArray[0] accède au premier élément du tableau, soit "Apple",
- L'instruction myArray[1] accède au deuxième élément, soit "Banana",
- L'instruction myArray[2] accède au troisième élément, soit "Orange".

Modifier un élément via son indice
Les indices sont également utilisés lorsqu'on vous souhaitez modifier un élément en luis attribuant une nouvelle valeur.

Exemple :

```
    let myArray = ["Apple", "Banana", "Orange"];
    myArray[1] = "Grapes"; // Remplace "Banana" par "Grapes"
       console.log(myArray);    // Affiche ["Apple", "Grapes",
"Orange"]
```

Dans cet exemple, l'élément à l'indice 1 (qui était "Banana") a été remplacé par "Grapes".

Accéder à la longueur d'un tableau
JavaScript vous fourni un attribut très simple pour connaitre la longueur d'un tableau (c'est-à-dire le nombre total d'éléments qu'il contient). Cet attribut est la propriété **lenght**. En fait, connaitre la longueur d'un tableau est nécessaire lorsqu'on souhaite parcourir les éléments d'un tableau, car l'indice du dernier élément est toujours égal à la longueur du tableau moins 1.

Exemple :

```
    let myArray = ["Apple", "Banana", "Orange"];
     console.log(myArray.length);   // Affiche 3, car il y a 3
éléments
```

Lorsque vous exécutez ce code, il sera affiché 3 (puisque le tableau possède 3 éléments)

Parcourir un tableau avec une boucle
Pour accéder aux éléments d'un tableau, de manière répétée, on utilise souvent les boucles comme le montre l'exemple ci-dessous :

Exemple :

```
let myArray = ["Apple", "Banana", "Orange"];
for (let i = 0; i < myArray.length; i++) {
    console.log(myArray[i]);
}
```

Dans cet exemple, la boucle **for** parcourt tous les éléments du tableau à l'aide du compteur i. À chaque itération, l'indice i est incrémenté de 1 et la valeur correspondante est récupérée dans la console du navigateur.

En conséquence, l'exécution du code ressemblera à:

```
Apple
Banana
Orange
```

Accéder au dernier élément du tableau

Si vous avez besoin d'accéder aux éléments en commençant par le dernier élément, vous pourriez utiliser des indices négatifs ou simplement soustraire des indices du nombre total d'éléments du tableau.

Exemple :

```
let myArray = ["Apple", "Banana", "Orange"];
   console.log(myArray[myArray.length - 1]);    // Affiche
"Orange", le dernier élément
```

Explication :

- La valeur myArray.length renvoie le nombre d'éléments du tableau, qui est ici 3.
- Pour accéder au dernier élément, nous utilisons l'indice myArray.length - 1.
- Puisque les indices JavaScript du tableau commence de

0, l'indice du dernier élément de tout tableau est toujours la longueur du tableau moins 1.

- Donc, myArray.length - 1 est égal à 3-1 = 2. Cela signifie que nous choisissons l'élément à l'index 2, qui est "Orange".

Exemples de fonctions/méthodes spécifiques aux tableaux

Dans cette section, nous vous montrerons certaines des méthodes/ fonctions les plus couramment utilisées pour manipuler les tableaux. Ces méthodes/fonctions sont utilisées pour ajouter, modifier, supprimer et transformer des éléments dans un tableau. Notez que nous utilisons spécifiquement le terme "**méthode**" au lieu de "**fonction**". Vous comprendrez pourquoi lorsque nous aborderons la programmation orientée objet avec JavaScript. Pour l'instant, considérez ces méthodes comme des fonctions qui agissent sur les tableaux. Voici les opérations les plus couramment utilisées :

- **Inverser l'ordre des éléments**

la méthode **reverse()** est utilisée pour inverser les éléments dans un tableau. Cela retourne le tableau d'origine avec un nouvel ordre d'élément.

Exemple :

```
    let fruits = ["Apple", "Banana", "Orange"];
    fruits.reverse();
      console.log(fruits);    // Affiche ["Orange", "Banana",
"Apple"]
```

Résultat :

```
['Orange', 'Banana', 'Apple']
```

Explication : Vous remarquez que nous avons obtenu un tableau constitué des mêmes éléments, mais en inversant leur ordre. Ici, "Apple" devient le dernier élément et "Orange" le premier.

- **Conversion en une chaîne**

La méthode **join()** a pour rôle de convertir un tableau en une chaine de caractères. Les éléments du tableau seront concaténés en une seule chaîne de caractères, séparés par le caractère spécifié en argument de la méthode join().

Exemple :

```
    let fruits = ["Apple", "Banana", "Orange"];
    let fruitString = fruits.join(", ");
      console.log(fruitString);    // Affiche "Apple, Banana, Orange"
```

Résultat :

```
Apple, Banana, Orange
```

Explication : Dans cet exemple, remarquez bien que le tableau fruitString a été convertit en une chaine de caractères constituée de ses éléments séparés par une virgule et un espace.

- **Convertir une chaîne en tableau**

Si vous voulez découper une chaîne de caractères et la transformer en un tableau, JavaScript vous fourni une méthode simple nommée **split().**

Exemple :

```
    let sentence = "I love JavaScript";
    let words = sentence.split(" ");
          console.log(words);      // Affiche ["I", "love", "JavaScript"]
```

Résultat:

```
['I', 'love', 'JavaScript']
```

Explication : La méthode **split(" ")** découpe la chaîne sentence en un tableau de mots, en utilisant l'espace comme séparateur.

- **Ajouter des éléments à la fin d'un tableau**

Pour ajouter un ou plusieurs éléments à la fin d'un tableau, JavaScript vous fournit la méthode **push().** Cette méthode renvoie également la nouvelle longueur du tableau obtenu.

Exemple :

```
    let fruits = ["Apple", "Banana"];
    fruits.push("Orange");
       console.log(fruits);    // Affiche ["Apple", "Banana",
"Orange"]
```

Explication : La méthode push("Orange") ajoute "Orange" à la fin du tableau fruits.

- **Supprimer le dernier élément d'un tableau**

En appliquant la méthode **pop()** à un tableau donné, le dernier élément de ce tableau sera supprimé.

Exemple :

```
    let fruits = ["Apple", "Banana", "Orange"];
    let lastFruit = fruits.pop();
    console.log(lastFruit);  // Affiche "Orange"
    console.log(fruits);  // Affiche ["Apple", "Banana"]
```

Résultat :

```
Orange
▶ (2) ['Apple', 'Banana']
```

Explication : Le résultat montre bien qu'en appliquant la méthode pop() au tableau fruits, le dernier élément "Orange" a été supprimé.

- **Ajouter des éléments au début du tableau**

Pour ajouter un ou plusieurs éléments au début d'un tableau, il existe une méthode nommé **unshift().**

Exemple :

```
    let fruits = ["Banana", "Orange"];
```

```
    fruits.unshift("Apple");
       console.log(fruits);    // Affiche ["Apple", "Banana",
"Orange"]
```

Résultat :

```
▶ (3) ['Apple', 'Banana', 'Orange']
```

Explication : La méthode **unshift("Apple")** ajoute "Apple" au début du tableau fruits.

- **Supprimer le premier élément d'un tableau**

La méthode **shift()** supprime le premier élément d'un tableau et renvoie cet élément.

Exemple :

```
    let fruits = ["Apple", "Banana", "Orange"];
    let firstFruit = fruits.shift();
    console.log(firstFruit);  // Affiche "Apple"
    console.log(fruits);  // Affiche ["Banana", "Orange"]
```

Résultat :

```
Apple
▶ (2) ['Banana', 'Orange']
```

Explication : La méthode **shift()** le premier élément du tableau fruits à savoir "Apple", et le renvoie. Le tableau ne contient plus que "Banana" et "Orange".

- **Remplacer des éléments d'un tableau par d'autres**

La méthode **splice()** est très puissante; elle est utilisée pour supprimer, remplacer ou ajouter des éléments à n'importe quelle position dans le tableau. Cette méthode prend de nombreux arguments, dont l'indice de départ et le nombre d'éléments à remplacer ou à supprimer.

Exemple :

```
    let fruits = ["Apple", "Banana", "Orange"];
    fruits.splice(1, 1, "Grapes");
       console.log(fruits);    // Affiche ["Apple", "Grapes",
"Orange"]
```

Résultat :

```
['Apple', 'Grapes', 'Orange']
```

Explication : Ici, **splice(1, 1, "Grapes")** remplace l'élément ayant l'indice 1 ("Banana") par "Grapes". Le deuxième argument 1 indique qu'un seul élément est supprimé.

- **Fusionner (concaténer) des tableaux**

La méthode concat() qui est utilisée pour fusionner deux ou plusieurs tableaux en un seul, tout en ne modifiant pas les tableaux d'origine.

Exemple :

```
    let fruits = ["Apple", "Banana"];
    let vegetables = ["Carrot", "Potato"];
    let food = fruits.concat(vegetables);
        console.log(food);    // Affiche ["Apple", "Banana",
"Carrot", "Potato"]
```

Résultat:

```
['Apple', 'Banana', 'Carrot', 'Potato']
```

Explication : En exécutant le code, vous remarquez que la méthode **concat()** a fusionné les deux tableaux en un seul tout en conservant tous les éléments.

- **Vérifier la présence d'un élément : La méthode includes()**

Une autre méthode fournie par JavaScript consiste à vérifier si un élément appartient à un tableau donné. C'est la méthode includes(). Elle renvoie true si l'élément est trouvé, sinon false.

Exemple :

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits.includes("Banana"));  // Affiche true
console.log(fruits.includes("Grapes"));  // Affiche false
```

Résultat :

```
true
false
```

Explication : Dans ce code, la méthode **includes()** vérifie si “Banana” est dans le tableau fruits et renvoie **true**, puisque banana appartient à ce tableau.
Dans la dernière ligne de code la méthode **includes** vérifie si Grapes est dans le tableau, le résultat renvoyé est false puisque Grapes n’appartient pas à ce tableau.

En guise de conclusion, ces méthodes peuvent être utilisées pour manipuler les tableaux en langage de programmation javascript. Si vous devez ajouter, supprimer, remplacer et vérifier l’existence des éléments, ces fonctions sont très utiles pour ce type de tâches. Vous pouvez dorénavant adopter ces méthodes pour vos scripts afin de facilement développer vos collections de données.

Tableaux multidimensionnels en JavaScript

Un tableau multidimensionnel en JavaScript est essentiellement un tableau de tableaux. Autrement dit, un tableau peut contenir d’autres tableaux en tant qu’éléments, ce qui nous permet de créer des structures plus complexes telles que des matrices ou des grilles de données.

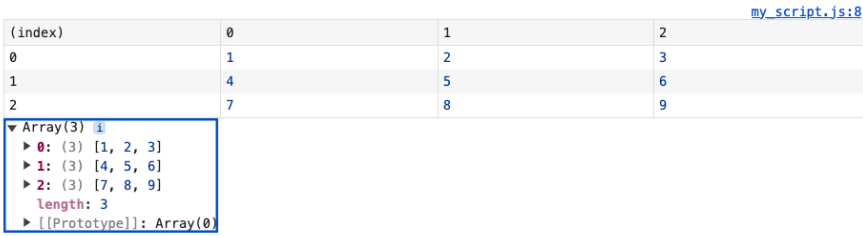
Pour créer un tableau multidimensionnel, nous devons imbriquer des tableaux les uns dans les autres. Par exemple, le tableau suivant pourrait être considéré comme une matrice de nombres à deux dimensions.

Exemple :

```
let matrix = [
```

```
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

console.log(matrix);
console.table(matrix);
```

Lorsque vous exécutez ce code, vous obtiendrez le résultat suivant :



Explication :

Le fait que le tableau matrix contient trois éléments et que chacun de ces éléments est lui-même un tableau contenant trois nombres signifie que :
L'élément d'indice 0 est le tableau [1,2,3],
L'élément d'indice 1 est le tableau [4,5,6],
L'élément d'indice 2 est le tableau [7,8,9]

Chacun des sous-tableaux indique une ligne d'une matrice, et chaque nombre dans ces sous-tableaux est accessible en utilisant son indice dans le tableau parent. Dans ce qui suit, nous allons voir comment accéder à ces éléments.

Accéder aux éléments d'un tableau multidimensionnel

Pour accéder à un élément spécifique d'un tableau multidimensionnel, vous utilisez deux indices: le premier pour accéder à l'élément du tableau "extérieur" et le second pour accéder aux sous-éléments du tableau "intérieur."

Exemple :

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
```

```
        [7, 8, 9]
    ];

    console.log(matrix[0][1]);  // ligne d'indice 0 et colonne
d'indice 1. Affiche 2
    console.log(matrix[2][0]);  // ligne d'indice 2 et colonne
d'indice 0. Affiche 7
```

Explication :

- matrix[0][1] permet d'accéder à l'élément ayant l'indice 1 du premier sous-tableau (d'indice 0), ce qui renvoie la valeur 2.
- matrix[2][0] permet d'accéder à l'élément ayant l'indice 0 du troisième sous-tableau (d'indice 2), ce qui renvoie la valeur 9.

Modifier un élément d'un tableau multidimensionnel
Vous pouvez également modifier les éléments dans un tableau multidimensionnel en utilisant leurs indices. Pour ce faire il suffit de spécifier les nouvelles valeurs.

Exemple :

```
    let matrix = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ];

     matrix[1][2] = 100;  // Remplace l'élément à la position
[1][2] par 100
    console.table(matrix);
```

Résultat de l'exécution du code :

my_script.js:8

(index)	0	1	2
0	1	2	3
1	4	5	100
2	7	8	9

```
▼ Array(3) i
  ▶ 0: (3) [1, 2, 3]
  ▶ 1: (3) [4, 5, 100]
  ▶ 2: (3) [7, 8, 9]
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Explication :

Comme le montre le résultat, ici, **matrix[1] [2]** renvoie la valeur 6, et nous la remplaçons par 100. La nouvelle matrice est alors :

```
[1, 2, 3]
[4, 5, 100]
[7, 8, 9]
```

Parcourir un tableau multidimensionnel
Pour parcourir un tableau multidimensionnel, vous pouvez utiliser des boucles imbriquées. Une boucle va parcourir le tableau extérieur, et une autre boucle va parcourir chaque sous-tableau.

Exemple :

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
        console.log(matrix[i][j]);
    }
}
```

Explication détaillée :

- **Première étape: initialiser le tableau matrix.**

Matrix est un tableau à deux dimensions contenant une matrice 3 × 3; chaque dimension contient trois sous-tableaux:

- Un sous-tableau sélectionné est [1, 2, 3],

- Un sous-tableau sélectionné est [4, 5, 6],
- Un sous-tableau sélectionné est [7, 8, 9],

Les trois sous-tableaux constituent les trois lignes de la matrice.

- **Deuxième étape : Boucle (for) sur les lignes:**

La boucle for est definite comme suit: for(let i = 0; i < matrix.length; i++) :
L'indice i commence de 0 à matrix.length (soit de 0 à 2 dans cet exemple)

- Quand i= 0, la première ligne [1, 2, 3],
- Quand i=1, la deuxième ligne [4, 5, 6],
- Et quand i=2, la troisième ligne [7, 8, 9] sont sélectionnées.

- **Troisième étape : Boucle (for) sur les colonnes de chaque ligne:**

Chaque sous-tableau une autre boucle for, for (let j = 0; j < matrix[i].length; i++) :
La variable j est l'indice qui parcourt chaque élément du sous-tableau. Par exemple :

- Quand i = 0 (première ligne), j parcourt les valeurs de [1, 2, 3], donc matrix[0][0], matrix[0][1], et matrix[0][2].
- Quand i = 1 (deuxième ligne), j parcourt les valeurs de [4, 5, 6].
- Quand i = 2 (troisième ligne), j parcourt les valeurs de [7, 8, 9].

- · Dernière étape : Afficher les éléments avec console.log(matrix[i][j]):

À chaque itération de la boucle intérieure, matrix[i][j] est utilisé pour accéder à un élément spécifique dans la matrice, et cet élément est affiché avec console.log().

Tableaux multidimensionnels de plus de deux dimensions

En JavaScript, vous pouvez également créer des tableaux de plus de deux dimensions, tels que des tableaux de trois dimensions ou plus, en incluant toujours plus de tableaux. Dans ce cas, chaque élément de ce tableau multidimensionnel est lui-même un tableau multidimensionnel.

Conclusion

En guise de conclusion de ce chapitre sur les tableaux en JavaScript, nous avons exploré un outil puissant et flexible pour gérer des collections de données. Les tableaux offrent de nombreuses possibilités, de l'organisation simple de valeurs à l'application de méthodes avancées pour manipuler et transformer les données. Que ce soit pour stocker des éléments, itérer sur des collections, ou appliquer des filtres et des transformations, les tableaux sont essentiels pour donner vie à vos projets JavaScript.

Les connaissances acquises ici sont une base solide pour aborder des concepts plus complexes, comme les objets et les structures de données imbriquées. Maîtriser les tableaux est un atout indéniable pour tout développeur souhaitant créer des applications web dynamiques et interactives.

Prêt pour la suite ? Préparez-vous à découvrir encore plus d’outils pour enrichir vos compétences en JavaScript !

Chapitre 8

Les objets en JavaScript

Les objets en JavaScript sont les structures de données qui permettent de stocker et d'organiser des informations complexes. Contrairement aux valeurs primitives telles que les nombres ou les chaînes de caractères, les objets sont des collections à clé et à valeur, ce qui leur permet de représenter des entités du monde réel telles qu'une personne, une voiture ou un produit, avec des caractéristiques et des actions.

JavaScript est plus particulièrement puissant lorsqu'il s'agit de modéliser et de manipuler des données grâce à des objets. Chaque propriété d'un objet est une paire clé-valeur, tandis que chaque fonction d'objet est une méthode (c'est-à-dire une fonction) de l'objet qui lui permet d'effectuer une action.

Ce chapitre vous introduira en profondeur à la création, à la manipulation et à l'utilisation des objets en JavaScript. Au cours de ce chapitre, vous apprendrez à définir un objet, à accéder à ses propriétés et à les modifier, à ajouter des méthodes à vos objets, et comment les utiliser pour rendre vos programmes JavaScript plus structurés. Les objets sont également le fondement de la programmation orientée objet, à laquelle nous nous intéresserons plus attentivement au cours des chapitres suivants.

Les objets en JavaScript : c'est quoi ?

Depuis le début de ce livre, nous avons manipulé plusieurs variables et avons bien compris qu'une variable JavaScript est un conteneur qui stocke des valeurs de données simples.

Exemples de variables simples :

```
let lastname = "Pascale";
let adress = "Rue de la République, numéro 123";
let score = 17;
```

Cependant, dans bien des cas, il nous faut gérer des collections de données plus complexes ou des entités qui regroupent plusieurs informations liées entre elles. C'est là qu'interviennent les objets en JavaScript. Un objet est une structure permettant de stocker des données variées sous forme de propriétés et de méthodes.
Les propriétés sont des variables définissant les caractéristiques d'un objet, alors que les méthodes sont des fonctions permettant de manipuler ces propriétés ou d'agir sur l'objet lui-même. En effet, en JavaScript, presque tout est un objet, comme la plupart des structures de données et des fonctionnalités que nous utilisons, tels que les tableaux, les fonctions, ou même les chaînes de caractères.

En outre, vous pouvez créer vos propres objets avec des propriétés et des méthodes adaptées à vos besoins. JavaScript étant un langage orienté objet, il est essentiel de comprendre ce concept afin de pouvoir progresser vers les sujets plus avancés du langage. C'est pourquoi cette partie vous fera découvrir les bases des objets en JavaScript avant d'approfondir la programmation orientée objets, qui constitue une base fondamentale de ce langage.

Objets littéraux

Comment créer un objet en JavaScript ?
En JavaScript, il existe de nombreuses façons de créer des objets, et l'une des plus simples et des plus utilisées est la notation de l'objet littéral. Il permet de déclarer rapidement un objet avec des propriétés et des méthodes à l'aide d'une syntaxe compacte et lisible.

Définition d'un objet littéral
Un objet littéral est un objet défini avec un ensemble de paires clé-valeur, chaque clé étant une propriété ou une méthode et chaque

valeur étant affectée à cette clé : utilisez les accolades {} pour définir un objet littéral. Nous utilisons les objets tous les jours et il y a une infinité d'objets dans la vie. Si je prends la voiture de mon fils comme exemple, il est certain qu'elle a des propriétés. Par exemple, elle a une marque, un modèle, une année de production, une couleur, et un nombre de portes. Pour chaque type de données et en fonction de ces données, nous allons créer un objet que je vais appeler en JavaScript par la suite.
Exemple de création d'objet littéral :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013,
    color: "Black",
    numberOfDoors: 5
};
```

Dans cet exemple, nous avons créé un **objet** appelé **car** avec **5 propriétés** : brand, model, year, color, et numberOfDoors.

Remarque :
Il est une bonne pratique de déclarer des objets en JavaScript en utilisant le mot clé **const**. Cela garantit que la référence à l'objet ne peut pas être changée une fois qu'elle est initialisée. Cela aide à prévenir les erreurs potentielles et rend votre code plus lisible et plus facile à entretenir. N'oubliez pas que même si l'objet est déclaré avec **const**, vous pouvez toujours changer les propriétés de l'objet, par exemple, vous pouvez changer la couleur ou le nombre de portes de l'objet « car ».
Si vous utilisiez **let** pour déclarer l'objet **car**, vous pourriez ensuite changer cet objet complète avec un autre objet et cela pourrait créer des erreurs ou des incohérences dans votre code. C'est la raison pour laquelle il est recommandé d'utiliser **const** pour déclarer des objets lorsqu'on sait qu'ils ne seront pas remplacés.

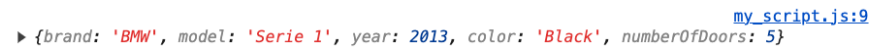
Affichons maintenant notre objet « **car** » dans la console :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013,
```

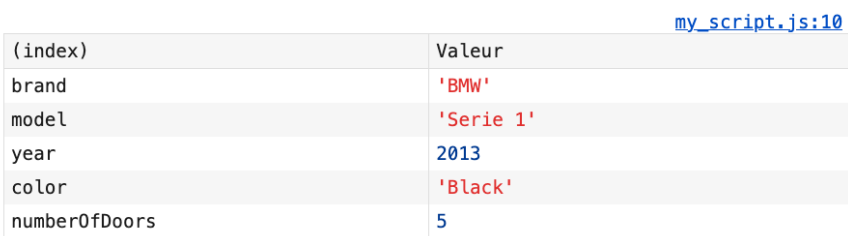
```
    color: "Black",
    numberOfDoors: 5
};

console.log(car);
console.table(car);
```

Explication :

L'instruction **console.log(car)** affiche l'objet **car** dans la console de votre navigateur. Vous obtiendrez l'objet avec toutes ses propriétés et leurs valeurs.



L'instruction **console.table(car)** affiche l'objet **car** sous la forme d'un tableau (ou d'une grille) dans la console. Chaque propriété de l'objet sera affichée dans une ligne avec deux colonnes.
Une colonne pour le nom de la propriété (par exemple, brand, model, year, etc.),
Une autre colonne pour la valeur correspondante (par exemple, BMW, Serie 1, 2013, etc.).



Propriétés d'un objet

Chaque **propriété** d'un objet n'est qu'une association entre un **nom de propriété** (qui est une chaîne) et une **valeur** (qui peut être de n'importe quel type de données, y compris un autre objet).

En exécutant l'exemple précédent, vous pouvez observer dans la console que sous l'onglet « Objet », chaque propriété est représentée par son nom, suivi d'un deux-points (:), et de sa valeur. Cette notation permet de structurer et d'organiser des données de manière claire et accessible, rendant chaque propriété de l'objet identifiable et

manipulable.

Ainsi, les propriétés d'un objet constituent la base de la gestion des données dans la programmation JavaScript.

NB : Il n'y a pratiquement aucune restriction concernant le nom des propriétés d'un objet. Il peut s'agir de n'importe quelle chaîne de caractères ou même d'un symbole (qui est un type spécial pour les identifiants, que nous aborderons plus tard). En fait, tout autre type de donnée sera automatiquement converti en chaîne de caractères.

Par exemple, si vous nommez une propriété avec un nombre comme 10, JavaScript le convertira automatiquement en une chaîne de caractères "10".

Accéder aux propriétés de l'objet
En JavaScript, vous pouvez accéder aux propriétés d'un objet de deux manières principales : à l'aide de la notation par point (.) ou de la notation par crochets ([]).

Notation par point :
C'est la méthode la plus courante et la plus simple pour accéder aux propriétés d'un objet. Vous écrivez le nom de l'objet, suivi d'un point (.), puis du nom de la propriété à laquelle vous voulez accéder.

Exemple :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

console.log(car.brand);  // Affiche "BMW"
console.log(car.year);   // Affiche 2013
```

Dans cet exemple, nous accédons à la propriété **brand** de l'objet **car** en utilisant la notation par point (.), en écrivant : **car.brand**. De la même manière, nous accédons à la propriété **year** de l'objet **car** en écrivant : **car.year**. Ici, le nom de la propriété est directement lié à l'objet, ce qui rend l'accès simple et intuitif.

Lorsque vous exécutez ce code, les valeurs des deux propriétés brand et year seront affichées grâce aux syntaxes car.brand et car.year. Cela permet d'extraire facilement les informations stockées dans l'objet.

```
BMW
2013
```

Notation par crochets :
La notation par crochets est particulièrement utile lorsque le nom de la propriété est dynamique (déterminé à l'exécution) ou lorsqu'il contient des caractères spéciaux ou des espaces. Dans ce cas, vous placez le nom de la propriété entre crochets et guillemets ("").

Exemple :

```
const car = {
    brand: "BMW",
    "number of doors": 5
};

console.log(car["brand"]);  // Affiche "BMW"
console.log(car["number of doors"]);  // Affiche 5
```

Dans cet exemple, nous accédons à la propriété « brand » en utilisant des crochets, ce qui donne le même résultat que la notation par point. Nous accédons également à la propriété « number of doors » en utilisant des crochets, car cette propriété contient des espaces, ce qui n'est pas autorisé avec la notation par point.

Quand utiliser chaque notation :

Notation par point : Utilisez cette méthode lorsque vous connaissez le nom exact de la propriété et que celui-ci ne contient ni espaces ni caractères spéciaux. C'est la manière la plus simple et directe d'accéder

à une propriété.

Notation par crochets : Privilégiez cette méthode lorsque le nom de la propriété est dynamique, qu'il contient des caractères spéciaux (comme des espaces) ou si vous devez accéder à la propriété à l'aide d'une variable.

Ces deux méthodes vous permettent de manipuler efficacement les propriétés d'un objet JavaScript en fonction de vos besoins.

Ajouter de nouvelles propriétés à un objet
En JavaScript, il est possible d'ajouter de nouvelles propriétés à un objet à tout moment, même après sa création initiale. Vous pouvez le faire soit en utilisant la notation par point, soit en utilisant la notation par crochets.

Utilisation de la notation par point :
La méthode la plus simple consiste à utiliser la notation par point pour ajouter une nouvelle propriété à un objet déjà existant. Vous écrivez simplement le nom de l'objet, suivi d'un point, puis du nom de la nouvelle propriété que vous souhaitez ajouter, et vous lui affectez une valeur.

Exemple :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

// Ajout d'une nouvelle propriété 'color' à l'objet 'car'
car.color = "Black";

console.log(car.color);
```

Dans cet exemple, nous avons utilisé la syntaxe par point (.) pour ajouter la propriété **color** à l'objet car avec la valeur "Black". Ensuite, nous avons utilisé console.table(car) pour afficher les propriétés et leurs valeurs de l'objet « car » dans un tableau, ainsi vérifier que la nouvelle propriété a bien été ajoutée.

Résultat :

my_script.js:10

(index)	Valeur
brand	'BMW'
model	'Serie 1'
year	2013
color	'Black'

▶ Object

Utilisation de la notation par crochets :
La notation par crochets est utile lorsque le nom de la nouvelle propriété est dynamique ou contient des caractères spéciaux, comme des espaces. Dans ce cas, vous placez le nom de la propriété entre crochets et guillemets.

Exemple :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

// Ajout d'une nouvelle propriété avec des espaces
car["number of doors"] = 5;

console.table(car);
```

Dans cet exemple, nous avons ajouté une propriété appelée number of doors (qui contient des espaces) à l'objet car. Grâce à la notation par crochets, nous pouvons gérer des noms de propriétés plus complexes.
L'exécution de ce code vous permet de vérifier que la propriété « number of doors » et sa valeur 5 ont été ajoués.

En conclusion, Vous pouvez modifier et enrichir vos objets en ajoutant des propriétés à tout moment.
Notation par point : Utilisée pour des noms de propriétés simples.
Notation par crochets : Préférée pour des noms dynamiques ou contenant des caractères spéciaux.
En utilisant l'une ou l'autre méthode, vous avez une grande flexibilité pour travailler avec des objets JavaScript et les adapter en fonction de vos besoins au fur et à mesure du développement de votre code.

Supprimer une propriété d'un objet

En JavaScript, il est également possible de supprimer une propriété d'un objet. Pour cela, on utilise l'opérateur **delete**. Cet opérateur permet de retirer une propriété et sa valeur de l'objet, laissant l'objet sans cette propriété.

Exemple avec la notation par point :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013,
    color: "Black"
};

// Suppression de la propriété 'color'
delete car.color;

console.log(car);  // La propriété 'color' a été supprimée
```

Dans cet exemple, la propriété **color** est supprimée de l'objet **car**. Après suppression, lorsque vous affichez l'objet, la propriété color n'apparaît plus dans l'objet.

```
{brand: 'BMW', model: 'Serie 1', year: 2013}
```

Exemple avec la notation par crochets :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013,
    "number of doors": 5
};

// Suppression de la propriété avec des espaces
delete car["number of doors"];

console.log(car);  // La propriété 'number of doors' a été supprimée
```

Résultat :

```
{brand: 'BMW', model: 'Serie 1', year: 2013}
```

Dans ce cas, nous avons utilisé la notation par crochets pour supprimer la propriété number of doors, qui contient des espaces. L'opération de suppression est réussie et la propriété n'existe plus dans l'objet.

À noter :

- La suppression d'une propriété à l'aide de l'opérateur delete retire complètement la clé (nom de la propriété) et sa valeur de l'objet.
- L'opérateur delete ne modifie pas la référence de l'objet lui-même. L'objet reste toujours accessible après la suppression de certaines de ses propriétés.
- Une fois qu'une propriété est supprimée, elle ne pourra plus être accédée (elle renverra undefined si vous essayez de l'utiliser).

L'utilisation de delete est donc une méthode efficace pour enlever des propriétés d'un objet sans modifier l'objet dans sa globalité.

Tester si une propriété existe dans un objet

En JavaScript, il peut être utile de vérifier si une propriété existe dans un objet avant d'essayer de l'utiliser. Il existe plusieurs façons de tester la présence d'une propriété dans un objet.

- **Utiliser l'opérateur in :**

L'opérateur in permet de tester si une propriété spécifique est présente dans un objet. Il renvoie true si la propriété existe, sinon il renvoie false.

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

console.log("brand" in car);  // Affiche: true
```

```
console.log("color" in car);  // Affiche: false
```

Dans cet exemple, "brand" in car renvoie true car la propriété brand existe dans l'objet car, tandis que "color" in car renvoie false car cette propriété n'existe pas dans l'objet.

- **Utiliser la méthode hasOwnProperty() :**

La méthode **hasOwnProperty()** est une autre façon de vérifier si une propriété existe dans un objet. Cette méthode renvoie également true ou false selon la présence ou non de la propriété.

Syntaxe :

```
object.hasOwnProperty("property");
```

Exemple :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

   console.log(car.hasOwnProperty("model"));    // Affiche: true
   console.log(car.hasOwnProperty("color"));    // Affiche: false
```

Dans cet exemple, **car.hasOwnProperty("model")** vérifie si l'objet « **car** » contient une propriété nommée « model » et renvoie **true** par ce que model est bien une propriété de car, tandis que **car.hasOwnProperty("color")** vérifie si l'objet **car** contient une propriété nommée **color** et dans ce cas, renvoie **false** par ce que cet cette propriété n'existe pas.

Différences entre in et hasOwnProperty() :

in : Cet opérateur vérifie si la propriété est présente dans l'objet ou dans sa chaîne de prototypes.

hasOwnProperty() : Cette méthode vérifie uniquement si la propriété

appartient directement à l'objet (elle ne teste pas les propriétés héritées du prototype).

Une autre méthode :

Lorsqu'une propriété n'existe pas dans un objet JavaScript, la tentative d'accès à cette propriété renvoie simplement la valeur undefined. Cela signifie que, même si vous essayez d'accéder à une propriété qui n'est pas définie dans un objet, JavaScript ne produira pas d'erreur ; au lieu de cela, il renverra undefined. Cette caractéristique permet de vérifier facilement si une propriété existe dans un objet en testant si sa valeur est undefined ou non.

Exemple :

```
    const car = {
        brand: "BMW",
        model: "Serie 1",
        year: 2013
    };

    console.log(car.year === undefined); // Affiche false, car
'year' est bien défini
```

Dans cet exemple, nous vérifions si la propriété **year** de l'objet **car** est définie en comparant sa valeur à **undefined** :

Ici, **car.year** est comparé à **undefined**. Comme la propriété year est définie dans l'objet et a la valeur 2013, la comparaison renverra false, indiquant que year est bien présent et défini dans l'objet.

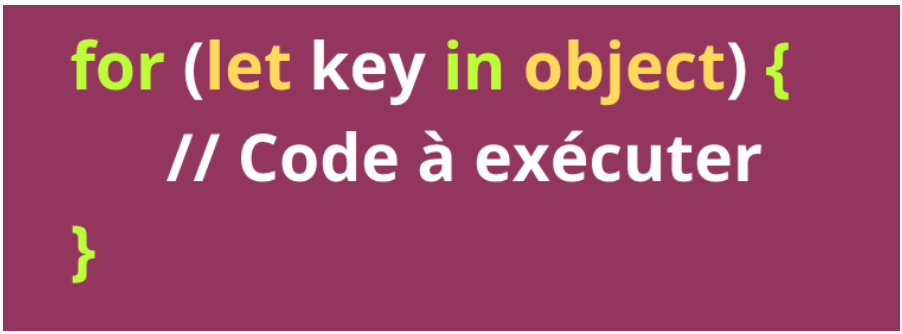
Limites de cette méthode :
Cette méthode fonctionne bien lorsque vous ne vous attendez pas à ce qu'une propriété ait explicitement la valeur undefined. Toutefois, si une propriété existante peut avoir cette valeur, cela pourrait poser problème, car le test renverrait alors **true** (c'est-à-dire que la propriété n'existe pas) même si la propriété est bien présente.

La boucle “for…in”

La boucle “for…in” est utilisée pour itérer sur les propriétés d'un objet en JavaScript. Elle permet de parcourir toutes les propriétés

énumérables de l'objet, qu'il s'agisse de propriétés définies directement sur l'objet lui-même ou héritées via la chaîne de prototypes.

Syntaxe :



- key représente le nom de la propriété actuelle à chaque itération.
- object est l'objet sur lequel vous itérez.

Exemple d'utilisation :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

for (let property in car) {
    console.log(property + ": " + car[property]);
}
```

L'exécution de ce code aboutit au résultat :

```
brand: BMW
model: Serie 1
year: 2013
```

Explication :
Dans cet exemple, la boucle "for…in" parcourt les propriétés de l'objet car. À chaque itération, la variable property contient le nom d'une propriété (par exemple, brand, model, year) et car[property] renvoie

la valeur associée à cette propriété.

Attention :
Il est important de noter que “**for…in**” itère également sur les propriétés héritées via le prototype, et non seulement sur les propriétés définies directement sur l'objet. Si vous voulez limiter la boucle uniquement aux propriétés "propres" à l'objet, vous pouvez utiliser hasOwnProperty() pour faire cette distinction.

Exemple avec hasOwnProperty() :

```
const car = {
    brand: "BMW",
    model: "Serie 1",
    year: 2013
};

for (let property in car) {
    if (car.hasOwnProperty(property)) {
        console.log(property + ": " + car[property]);
    }
}
```

Dans cette version, grace à la méthode `hasOwnProperty`, le code vérifiera que la propriété fait bien partie de l'objet « car » lui-même et non d'un prototype. Cela peut être utile pour éviter des comportements inattendus lorsque des propriétés sont héritées.

La boucle “for…in” est donc particulièrement adaptée pour parcourir les objets, permettant de manipuler leurs propriétés de manière dynamique.

Méthodes d'objet en JavaScript :

En JavaScript, une méthode est simplement une fonction associée à un objet. Ces méthodes permettent d'accomplir des actions spécifiques sur les propriétés de l'objet, ou d'interagir avec l'objet lui-même. Contrairement aux fonctions classiques qui sont définies indépendamment, une méthode est définie comme une propriété d'un objet, dont la valeur est une fonction.

Exemple :
Une méthode est définie de la même manière qu'une propriété, sauf que la valeur attribuée est une fonction.

```
const person = {
    firstName: "Karine",
    lastName: "Lambert",
    fullName: function () {
        return this.firstName + " " + this.lastName;
    }
};
```

Dans cet exemple, fullName est une méthode de l'objet person qui renvoie le nom complet de la personne en accédant à ses propriétés firstName et lastName via this. À ce stade, nous avons juste déclarer que fullName est une méthode.

Utilisation de la méthode :
Pour appeler une méthode, on utilise la même syntaxe que pour accéder à une propriété, en ajoutant des parenthèses () après le nom de la méthode. Ajoutons le code suivant :

```
console.log(person.fullName());  // Affiche "Karine Lambert"
```

Ensuite, vous exécutez le code obtenu.
Résultat :

```
Karine Lambert
```

Le mot-clé this
Dans l'exemple précédent, nous avons utilisé le mot-clé **this. Notez que ce mot-clé** dans une méthode fait référence à l'objet auquel la méthode appartient. Dans notre exemple, **this.firstName** et **this.lastName** font référence aux propriétés firstName et lastName de l'objet person.

Exemple d'objet avec plusieurs méthodes :

```
const car = {
    brand: "Tesla",
    model: "Model S",
    year: 2022,
```

```
    getCarInfo: function () {
        return this.brand + " " + this.model;
    },
    getCarYear: function () {
        return this.year;
    }
};

console.log(car.getCarInfo()); // Affiche "Tesla Model S"
console.log(car.getCarYear());  // Affiche 2022
```

Dans cet exemple, l'objet « **car** » possède deux méthodes : **getCarInfo** et **getCarYear**. La première méthode renvoie la marque et le modèle de la voiture, tandis que la seconde renvoie l'année de la voiture.

```
Tesla Model S
2022
```

Pourquoi utiliser des méthodes ?
Les méthodes dans les objets permettent d'encapsuler des fonctionnalités relatives à l'objet. Elles facilitent l'organisation et la gestion du code, rendant l'objet plus autonome et cohérent. Cela permet également de manipuler les données associées à l'objet de manière plus pratique.

Les méthodes d'objets sont fondamentales en JavaScript car elles permettent de donner du comportement aux objets, en plus de simplement stocker des données.

Constructeur d'objets en JavaScript

Dans JavaScript, les objets peuvent être créés de plusieurs manières. L'une des méthodes les plus efficaces et réutilisables consiste à utiliser des constructeurs d'objets. Un constructeur d'objet est simplement une fonction qui permet de créer plusieurs instances d'un même type d'objet, chacune avec ses propres propriétés et méthodes.

Pourquoi utiliser un constructeur d'objets ?
Les constructeurs sont particulièrement utiles lorsque vous devez créer plusieurs objets similaires, avec des propriétés et des méthodes similaires. Plutôt que de déclarer chaque objet manuellement, vous

pouvez créer une fonction constructeur qui vous permettra de générer des objets de manière dynamique et structurée.

Syntaxe du constructeur d'objets
Un constructeur d'objets en JavaScript est une fonction définie avec un nom commençant par une majuscule (par convention). À l'intérieur de cette fonction, le mot-clé **this** est utilisé pour assigner des propriétés spécifiques à chaque instance d'objet que vous allez créer.

Exemple de constructeur d'objet :

```
    function Car(brand, model, year, color) {
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.color = color;
        this.displayInfo = function () {
                return this.brand + " " + this.model + ", " +
this.year + ", " + this.color;
        };
    }
```

Dans cet exemple, nous avons créé une fonction **Car** qui agit comme un constructeur. Elle accepte plusieurs paramètres : brand (marque), model (modèle), year (année), et color (couleur). Le mot-clé **this** fait référence à l'objet qui sera créé par cette fonction. À chaque nouvel objet créé, ces propriétés auront des valeurs distinctes.

Création d'objets à partir du constructeur
Pour créer un nouvel objet à partir d'un constructeur, nous utilisons l'opérateur new. Cet opérateur instancie un nouvel objet à partir du constructeur spécifié.

Exemple d'utilisation :

```
    function Car(brand, model, year, color) {
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.color = color;
        this.displayInfo = function () {
                return this.brand + " " + this.model + ", " +
```

```
this.year + ", " + this.color;
        };
    }

    let car1 = new Car("BMW", "Serie 1", 2013, "Black");
    let car2 = new Car("Tesla", "Model S", 2022, "White");

    console.log(car1); // Affiche : BMW Serie 1, 2013, Black
    console.log(car2); // Affiche : Tesla Model S, 2022, White
```

Dans cet exemple, nous avons créé deux objets distincts, car1 et car2, en utilisant le même constructeur Car. La création d'un objet via un constructeur s'effectue à l'aide du mot-clé **new** suivi du nom du constructeur, avec les paramètres spécifiant les propriétés entre parenthèses. Ces deux objets possèdent la même structure, mais les valeurs de leurs propriétés sont différentes. De plus, ils partagent la méthode displayInfo(), qui renvoie une chaîne de caractères contenant les informations de l'objet.

Les deux dernières lignes du code permettent d'afficher les informations des objets car1 et car2. Si vous exécutez ce code, vous verrez le résultat suivant :

```
                                                    my_script.js:14
▶ Car {brand: 'BMW', model: 'Serie 1', year: 2013, color: 'Black', displayInfo: f}
                                                    my_script.js:15
▶ Car {brand: 'Tesla', model: 'Model S', year: 2022, color: 'White', displayInfo: f}
```

Avantages d'utiliser des constructeurs d'objets

Réutilisabilité : Plutôt que de recréer manuellement chaque objet, vous pouvez facilement générer plusieurs objets avec la même structure à partir du constructeur.
Clarté : L'utilisation d'un constructeur clarifie le processus de création d'objets et montre clairement les types d'objets que vous travaillez dans votre application.

Extensibilité : Les constructeurs peuvent être enrichis avec des méthodes et des propriétés supplémentaires au fil du temps, offrant une grande flexibilité.

Remarque sur le mot-clé this
Le mot-clé this fait référence à l'instance spécifique de l'objet que vous

créez. Lors de l'appel de la fonction constructeur avec new, this prend une nouvelle valeur à chaque fois, pointant vers l'objet en cours de création.

Remarque importante

Il est important de noter que ce que nous venons de voir correspond à la façon de créer des objets avant 2015, c'est-à-dire avant l'introduction de la version ES6 de JavaScript. Avec ES6, une notion cruciale a été introduite : celle des classes. Nous allons explorer cette notion en détail dans la suite de cette formation.

En guise de conclusion, les constructeurs d'objets sont un élément fondamental en programmation orientée objet dans JavaScript. Ils permettent de créer des objets dynamiques et réutilisables avec une structure claire et cohérente. Utiliser des constructeurs, c'est rendre le code plus flexible et facilement maintenable, surtout lorsque l'on doit manipuler des objets avec les mêmes caractéristiques.

Conclusion

Pour conclure ce chapitre sur les objets en JavaScript, nous avons découvert que les objets sont des structures fondamentales dans ce langage. Ils permettent de regrouper des données sous forme de propriétés et de méthodes, offrant ainsi un moyen puissant d'organiser et de manipuler des informations de manière structurée.

Que ce soit en utilisant des objets littéraux ou des constructeurs, nous avons vu comment créer et manipuler des objets, accéder à leurs propriétés, ajouter ou supprimer des attributs, et même vérifier leur existence. Ces concepts sont essentiels pour comprendre la puissance de JavaScript, notamment dans le cadre de la programmation orientée objet.

En maîtrisant les objets, vous vous préparez à aborder des concepts plus avancés comme les classes, les prototypes, et l'héritage, qui jouent un rôle majeur dans la conception de programmes robustes et modulaires en JavaScript. Nous allons justement aborder ces notions dans les chapitres suivants pour vous permettre d'approfondir vos compétences et de progresser vers une meilleure maîtrise du langage.

Chapitre 9

Classes et objets

L'un des fondements de la programmation orientée objet (POO) est la notion de classes et d'objets. Que ce soit pour modéliser une personne, un véhicule ou une application entière, les classes vous permettent de définir des modèles réutilisables qui représentent des entités du monde réel ou abstrait. Grâce aux classes, vous pouvez non seulement structurer votre code, mais aussi simplifier sa maintenance, sa réutilisabilité et son évolutivité.

En JavaScript, bien que les objets aient toujours été au cœur du langage, la syntaxe des classes a été introduite avec ES6, rendant la création d'objets plus intuitive et plus proche des autres langages de programmation orientée objet, tels que Python ou Java. Les classes apportent une manière plus claire de structurer vos programmes et facilitent l'organisation des concepts tels que l'héritage, les méthodes et les propriétés.

Dans ce chapitre, nous allons explorer :

- Ce qu'est une classe : Comment définir et utiliser une classe en JavaScript.
- Les objets : La création d'objets à partir de classes, et l'importance des propriétés et des méthodes.
- Le constructeur : La fonction spéciale qui initialise les objets avec des valeurs spécifiques.
- Méthodes d'une classe : Comment attacher des comportements (fonctions) à des objets.

À travers des exemples concrets, vous apprendrez comment les classes permettent de simplifier la manipulation de données complexes et d'organiser votre code de manière plus claire et modulaire. Vous verrez également comment utiliser ces concepts pour concevoir des applications évolutives et faciles à maintenir.

Ce chapitre est une étape essentielle dans votre apprentissage de la POO en JavaScript. Comprendre comment utiliser les classes et les objets efficacement vous ouvrira la porte à des possibilités infinies dans la conception de vos projets futurs. Plongeons dans ce monde où

chaque élément devient un objet, et où les classes sont la clé pour les créer.

Introduction aux classes

La programmation orientée objet (POO) est un paradigme essentiel qui permet de structurer et d'organiser le code de manière modulaire et réutilisable. L'un des concepts centraux de la POO est celui de classe et objet. Dans ce chapitre, nous allons explorer en profondeur comment JavaScript permet de créer et d'utiliser des classes et des objets, en nous concentrant sur les concepts de base et en apportant des exemples pratiques et détaillés.

En JavaScript, une classe est un plan ou un modèle qui permet de créer des objets. Chaque objet créé à partir d'une classe est une instance de cette classe. Les classes encapsulent des données sous forme de propriétés (variables) et des méthodes (fonctions), qui définissent les actions que l'objet peut effectuer.

Définition et création de classes

Les classes représentent le pilier central de la programmation orientée objet en JavaScript. Une classe définit la structure d'un objet en regroupant à la fois des données (appelées propriétés) et des comportements (appelés méthodes) qui sont propres à un type particulier d'objet. En d'autres termes, une classe sert de modèle qui encapsule à la fois les caractéristiques et les actions d'un ensemble d'objets.

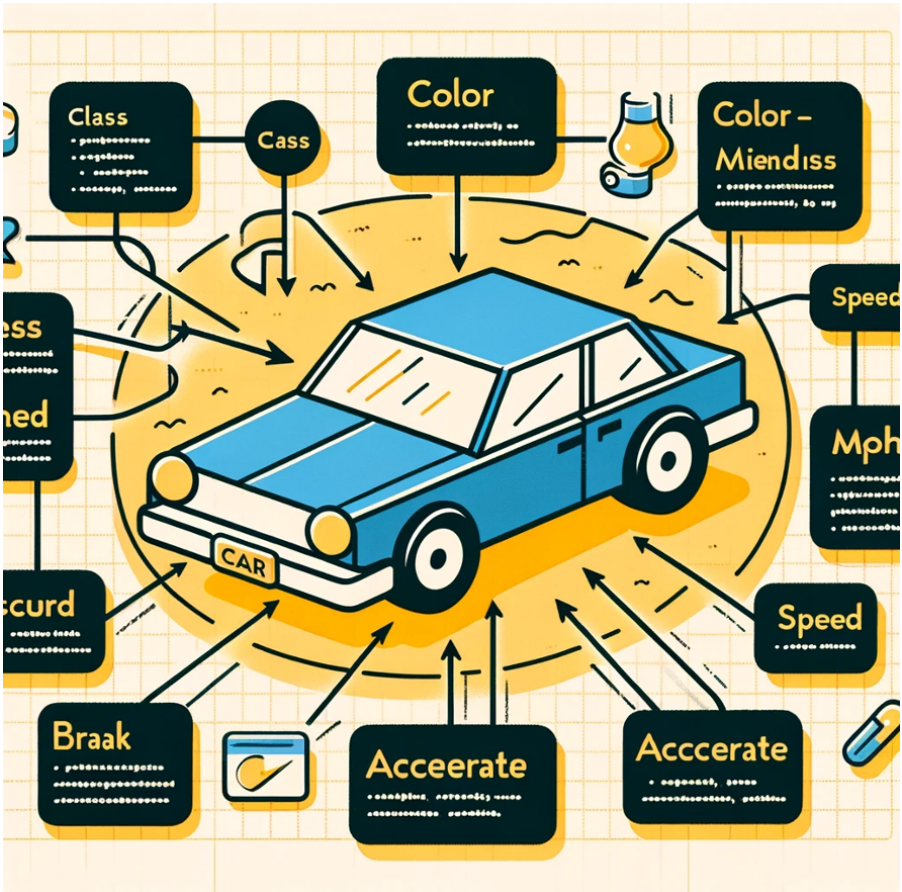
Pour mieux comprendre ce qu'est une classe, prenons une analogie simple :

Imaginez qu'une classe soit un plan architectural ou un modèle utilisé pour construire des maisons. Dans ce plan, vous définissez des caractéristiques communes que toutes les maisons partageront, comme le nombre de pièces, la disposition des pièces, les matériaux de construction, etc.

De la même manière, une classe en programmation est un modèle abstrait qui spécifie les caractéristiques (propriétés) et les comportements (méthodes) communs à un certain type d'objet. Par exemple, vous pourriez créer une classe "Voiture" (Car) qui contient des propriétés telles que la marque, le modèle, et la couleur, ainsi que des méthodes telles que démarrer, accélérer, et freiner.

Une fois la classe définie, vous pouvez créer des instances de cette classe. Chaque instance est un objet spécifique basé sur le modèle

défini par la classe. Par exemple, une instance de la classe "Voiture" serait une voiture particulière avec ses propres valeurs pour les propriétés telles que "Toyota", "Corolla", et "rouge".



En résumé, une classe est un modèle ou un plan qui définit la structure et le comportement d'un certain type d'objet, tandis qu'une instance est un exemplaire spécifique de cette classe, avec des valeurs propres à cet objet.

Définition et syntaxe d'une classe

En JavaScript, une classe est déclarée à l'aide du mot-clé **class**, suivi du nom de la classe et d'un bloc d'accolades **{}** qui encapsule ses propriétés et méthodes. Les classes permettent de regrouper des données et des fonctionnalités qui décrivent un objet particulier.

Le nom de la classe doit respecter certaines règles de syntaxe : il doit commencer par une lettre ou un trait de soulignement (_), et peut être

suivi de lettres, de chiffres, ou de traits de soulignement. Toutefois, par convention, il est recommandé d'utiliser la notation **PascalCase** pour nommer les classes. Cela signifie que chaque mot dans le nom de la classe commence par une majuscule et qu'il n'y a pas d'espaces ni de caractères spéciaux.

Par exemple, **Person**, **Car**, ou **UserCompte** suivent cette convention.

Il est important de noter que la première lettre du nom d'une classe est généralement en **majuscule**, selon les conventions de nommage en JavaScript. Cette pratique aide à distinguer facilement les classes des autres types de variables.

Voici la syntaxe générale de la définition d'une classe en JavaScript :

```
class NomClasse {
    // Contenu d'un classe
}
```

Comment créer (déclarer) une classe en JavaScript ?

Imaginons que nous souhaitions créer un espace membre pour gérer les informations de chaque utilisateur. Pour cela, il serait judicieux de définir une classe qui regroupe toutes les propriétés et méthodes relatives à une personne, afin de pouvoir instancier (créer) plusieurs objets représentant différentes personnes.

En JavaScript, il est de coutume de nommer les classes avec une première lettre en majuscule, bien que cela ne soit pas une obligation, mais une convention recommandée. La création d'une classe s'effectue à l'aide du mot-clé class.

- Les propriétés sont des variables qui stockent les caractéristiques de chaque objet (par exemple, le nom, l'âge, etc.).
- Les méthodes sont des fonctions définies au sein de la classe, qui représentent les actions que l'objet peut accomplir. Ces méthodes sont directement déclarées à l'intérieur de la classe, mais il n'est pas nécessaire d'utiliser le mot-clé function pour les déclarer.

Voici un exemple simple pour illustrer cela :
Dans votre répertoire de travail, créez un fichier nommé Person.js et insérez-y le code suivant :

```
class Person {
    constructor(name, age) {
        this.name = name; // propriété "name"
        this.age = age;   // propriété "age"
    }

    // Méthode pour dire bonjour
    sayHello() {
        console.log(`Bonjour, je m'appelle ${this.name}.`);
    }

    // Méthode pour afficher l'âge
    displayAge() {
        console.log(`J'ai ${this.age} ans.`);
    }
}
```

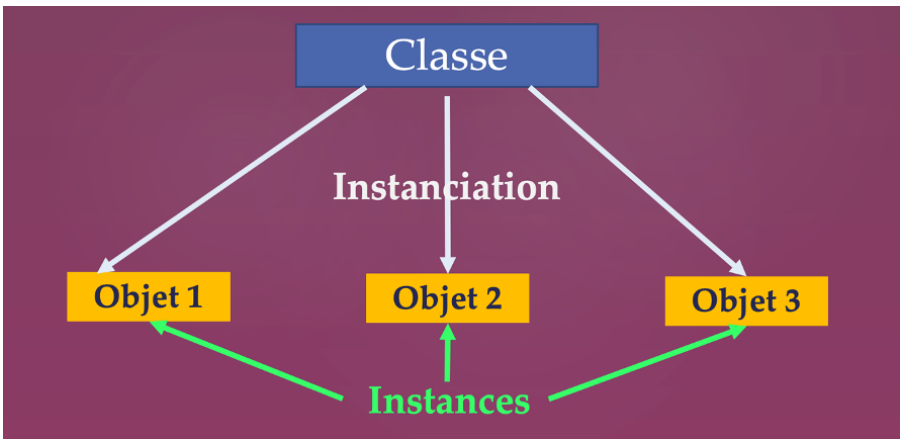
Explication :

- **class Person** : Ici, nous définissons une classe appelée Person.
- **constructor(name, age)** : Le constructeur est une méthode spéciale qui est appelée automatiquement lors de la création d'une nouvelle instance de la classe. Il initialise les propriétés de l'objet (name et age).
- **this.name** et **this.age** : Le mot-clé **this** fait référence à l'instance actuelle de la classe. Cela permet d'attacher des valeurs spécifiques à chaque instance.
- **Méthodes** : **sayHello()** et **displayAge()** sont des méthodes attachées à la classe. Elles définissent les actions que les objets créés à partir de cette classe peuvent exécuter.
-

À ce stade, bien que nous ayons créé la classe, si vous essayez d'exécuter le code, rien ne se passera. Pour rendre cette classe fonctionnelle, il est essentiel de l'utiliser en créant des objets à partir de celle-ci (en l'instanciant).

Instanciation d'objets

Une fois qu'une classe est définie, nous pouvons créer des objets (instances) de cette classe.
Pour créer une instance (un objet) d'une classe, on utilise le mot-clé **new**. Cela appelle automatiquement le constructeur de la classe.



Ajoutons le code suivant au fichier Person.js (à l'extérieur de la classe)

```
// Créer une instance de la classe Person
const person1 = new Person('Alice', 30);

// Utilisation des méthodes de l'objet
person1.sayHello(); // Bonjour, je m'appelle Alice.
person1.displayAge(); // J'ai 30 ans.
```

Dans la première ligne de ce code, nous utilisons le mot-clé **new** pour créer une instance de la classe Person. Une instance est un objet unique basé sur le modèle défini par la classe. Dans ce cas, la classe Person attend deux paramètres dans son constructeur : name (nom) et age (âge). En appelant new Person('Alice', 30), nous créons un nouvel objet qui représente une personne dont le nom est Alice et l'âge est de 30 ans.

- **new** : Ce mot-clé permet de créer une nouvelle instance de la classe.
- **Person('Alice', 30)** : Cela appelle le constructeur de la classe Person en lui passant les arguments 'Alice' et 30. Le constructeur initialise les propriétés name et age de cette nouvelle instance avec ces valeurs.

Une fois l'objet person1 créé, nous pouvons appeler ses méthodes. Ici, nous utilisons la méthode **sayHello()**, qui affiche un message de bienvenue dans la console.

- **person1.sayHello()** : Cette méthode fait partie de la classe Person, et elle accède à la propriété name de l'objet pour afficher le message Bonjour, je m'appelle Alice.. La propriété name a été initialisée avec la valeur 'Alice' lors de la création de l'instance.

Ensuite, nous utilisons la méthode **displayAge()**, qui affiche l'âge de l'objet person1.

- **person1.displayAge()** : Cette méthode accède à la propriété age de l'objet person1 et affiche le message J'ai 30 ans dans la console. La propriété age a été initialisée avec la valeur 30 lors de la création de l'instance.

Afin de tester votre code, nous allons créer une simple page HTML nommée index.html à laquelle nous allons lier le fichier Person.js.

Index.html

```
    <!DOCTYPE html>
    <html>

    <head>
        <meta charset="utf-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1">
        <title>Formation JavaScript</title>
    </head>

    <body>
        <h1 id="title1">Formation JavaScript</h1>
        <div class="introduction">
            <h2 id="intro">Introduction au DOM</h2>
             <p>Voici le paragrpahe 1 avec du texte introductif
1</p>
             <p>Voici le paragrpahe 2 avec du texte introductif
2</p>
             <p>Voici le paragrpahe 3 avec du texte introductif
```

```
3</p>
        </div>
        <script src="Person.js"></script>
    </body>

    </html>
```

Lorsque vous ouvrirez cette page, vous verrez apparaître dans la console de votre navigateur le résultat suivant :

```
Bonjour, je m'appelle Alice.
J'ai 30 ans.
```

Cet affichage correspond à l'exécution des deux méthodes **sayHello()** et **displayAge()** appliquées sur l'objet person1.
En résumé, ce code JavaScript que nous venons d'ajouter à Person.js montre comment instancier un objet à partir d'une classe et utiliser les méthodes sur cet objet. Chaque méthode peut accéder aux propriétés de l'objet (name, age) et effectuer des actions en conséquence, comme afficher des messages personnalisés.

Exemples supplémentaires :

Ajoutons également le code suivant à la fin du fichier Person.js :

```
const person2 = new Person('Bob', 25);
person2.sayHello(); // Bonjour, je m'appelle Bob.
person2.displayAge(); // J'ai 25 ans.
```

Ce code crée un nouveau objet (instance) nommé person2 de la classe Person et appelle les méthodes sayHello() et displayAge() sur cet objet.

Maintenant, lorsque vous ouvrirez cette page, vous verrez apparaître dans la console de votre navigateur le résultat suivant :

```
Bonjour, je m'appelle Alice.
J'ai 30 ans.
Bonjour, je m'appelle Bob.
J'ai 25 ans.
```

Nous avons donc **person1** et **person2** deux instances distinctes de la

classe Person. Elles partagent la même structure (les mêmes méthodes et propriétés), mais leurs valeurs sont spécifiques à chaque instance. Autrement dit, chaque objet créé à partir d'une classe aura ses propres valeurs pour les propriétés définies dans le constructeur.

Les propriétés d'une classe

Les propriétés d'une classe sont les variables qui définissent l'état d'un objet. Elles sont initialisées dans le constructeur, et chaque instance de la classe a ses propres valeurs pour ces propriétés. Il est également possible de modifier ou d'accéder à ces propriétés en dehors du constructeur.

```
  person1.name = 'Alice Dupont'; // Changer la valeur de la propriété "name"
 console.log(person1.name); // Alice Dupont
```

- Ici, la première ligne de ce code permet d'accèder à la propriété **name** de l'objet **person1** et on lui assigne une nouvelle valeur, "Alice Dupont". Si la propriété name avait déjà une valeur (par exemple "John"), celle-ci est écrasée par "Alice Dupont".
- C'est une manière de modifier dynamiquement une propriété d'un objet après sa création.
- La deuxième ligne utilise la fonction console.log() pour afficher la valeur actuelle de la propriété name de l'objet person1 dans la console. Comme cette propriété a été modifiée juste avant, le résultat affiché sera "Alice Dupont".

Valeurs par défaut des propriétés

Vous pouvez également définir des valeurs par défaut pour les propriétés dans le constructeur. Créez un nouveau fichier Car.js dans lequel vous mettez le code suivant pour créer la classe Car et ses instances car1 et car2 :

```
   class Car {
       constructor(brand = 'Toyota', model = 'Corolla') {
           this.brand = brand;
           this.model = model;
       }
```

```
        displayCar() {
                    console.log(`Marque: ${this.brand}, Modèle:
${this.model}`);
        }
    }

    const car1 = new Car(); // Utilise les valeurs par défaut
    car1.displayCar(); // Marque: Toyota, Modèle: Corolla

    const car2 = new Car('Honda', 'Civic');
    car2.displayCar(); // Marque: Honda, Modèle: Civic
```

Dans cet exemple, si vous ne fournissez pas de valeurs pour **brand** et **model**, les valeurs par défaut seront utilisées. « Toyota » pour la propriété brand et « Corolla » pour la propriété Model.

Les méthodes d'une classe

Les méthodes sont des fonctions attachées à une classe qui permettent aux objets d'exécuter des actions. Une méthode peut accéder aux propriétés de l'instance via le mot-clé this. Vous pouvez ajouter autant de méthodes que nécessaire pour définir les comportements de vos objets.

Voici un exemple plus complet :

Rectangle.js

```
    class Rectangle {
        constructor(width, height) {
            this.width = width;
            this.height = height;
        }

        // Méthode pour calculer l'aire du rectangle
        calculateArea() {
            return this.width * this.height;
        }
```

```
        // Méthode pour changer les dimensions
        setDimensions(newWidth, newHeight) {
            this.width = newWidth;
            this.height = newHeight;
        }
    }

    const rect1 = new Rectangle(10, 5);
    console.log(rect1.calculateArea()); // 50

    rect1.setDimensions(20, 10);
    console.log(rect1.calculateArea()); // 200
```

Explication détaillée:

Ce code JavaScript définit une classe appelée **Rectangle** qui permet de créer des objets représentant des rectangles avec des dimensions spécifiques (largeur et hauteur). Voici une explication détaillée des différentes parties du code :

1- Déclaration de la classe Rectangle

```
class Rectangle {
        constructor(width, height) {
            this.width = width;
            this.height = height;
        }
```

- La classe **Rectangle** est définie avec un constructeur. Le constructeur est une méthode spéciale utilisée pour initialiser les objets créés à partir de la classe.
- **constructor(width, height)** : Ce constructeur prend deux paramètres, width (largeur) et height (hauteur). Ces paramètres sont utilisés pour initialiser les propriétés de l'objet (this.width et this.height), qui représentent respectivement la largeur et la hauteur du rectangle.

2- Méthode calculateArea()

```
// Méthode pour calculer l'aire du rectangle
        calculateArea() {
            return this.width * this.height;
```

```
    }
```

- Cette méthode calcule et renvoie l'aire du rectangle.
- **this.width** et **this.height** font référence aux propriétés de l'objet Rectangle. L'aire est obtenue en multipliant la largeur par la hauteur.
- Par exemple, pour un rectangle avec une largeur de 10 et une hauteur de 5, l'aire sera 10 * 5 = 50.

3. Méthode setDimensions(newWidth, newHeight)

```
// Méthode pour changer les dimensions
    setDimensions(newWidth, newHeight) {
        this.width = newWidth;
        this.height = newHeight;
    }
```

- Cette méthode permet de modifier les dimensions du rectangle après sa création. Elle prend deux paramètres, **newWidth** et **newHeight**, qui correspondent à la nouvelle largeur et à la nouvelle hauteur.
- Les propriétés de l'objet **this.width** et **this.height** sont mises à jour avec ces nouvelles valeurs.

4. Création d'une instance de Rectangle

```
const rect1 = new Rectangle(10, 5);
```

Ici, un nouvel objet rect1 est créé à partir de la classe Rectangle. Les valeurs 10 et 5 sont passées au constructeur, ce qui initialise la largeur à 10 et la hauteur à 5 pour cet objet.

5.Calcul de l'aire du rectangle

```
console.log(rect1.calculateArea()); // 50
```

Cette ligne appelle la méthode **calculateArea()** sur l'objet rect1. L'aire calculée est 50 (puisque 10 * 5 = 50), et le résultat est affiché dans la console.

6. Modification des dimensions et recalcul de l'aire

```
rect1.setDimensions(20, 10);
    console.log(rect1.calculateArea()); // 200
```

- **La méthode setDimensions()** est utilisée pour modifier la largeur et la hauteur de rect1 en 20 et 10, respectivement.
- Ensuite, **calculateArea()** est appelé à nouveau pour calculer l'aire avec les nouvelles dimensions. L'aire est désormais 200 (puisque 20 * 10 = 200).

Conclusion :

- **Création d'objets dynamiques** : Ce code montre comment on peut créer des objets à partir d'une classe et leur attribuer des propriétés.
- **Encapsulation** : Il permet de cacher la complexité des calculs tout en offrant des méthodes simples comme **calculateArea()** et **setDimensions()** pour manipuler les objets.
- **Réutilisation et flexibilité** : Grâce à la classe Rectangle, on peut facilement créer plusieurs rectangles, calculer leurs aires et modifier leurs dimensions sans réécrire tout le code.

Accesseurs et mutateurs (getters et setters)

JavaScript permet également de créer des accesseurs (getters) et des mutateurs (setters) pour manipuler les propriétés d'un objet de manière contrôlée. Cela permet d'ajouter une logique supplémentaire lorsque les propriétés sont récupérées ou modifiées. Modifions la classe Person comme suit :

```
    class Person {
        constructor(name, age) {
            this._name = name; // Propriété privée
            this._age = age;
        }

        // Getter pour name
        get name() {
```

```
            return this._name.toUpperCase();
        }

        // Setter pour name
        set name(newName) {
            this._name = newName;
        }

        // Getter pour age
        get age() {
            return this._age;
        }

        // Setter pour age
        set age(newAge) {
            if (newAge > 0) {
                this._age = newAge;
            } else {
                console.log('Âge invalide');
            }
        }
    }

    const person3 = new Person('Lucie', 22);
      console.log(person3.name); // LUCIE (le getter modifie
l'affichage)
     person3.name = 'Emma'; // Utilise le setter pour changer
le nom
    console.log(person3.name); // EMMA

    person3.age = -5; // Affiche "Âge invalide"
```

Ce code utilise les getters et setters pour gérer l'accès aux propriétés d'un objet de manière contrôlée. Les getters permettent de récupérer des valeurs, tandis que les setters permettent de définir ou de modifier ces valeurs tout en appliquant des règles de validation si nécessaire. Voyons en détail comment cela fonctionne dans ce code.

1.constructor(name, age) : Cette méthode est le constructeur de la

classe. Elle initialise deux propriétés : **_name** et **_age**, qui sont ici écrites avec un **underscore (_)** au début pour indiquer que ce sont des propriétés internes (ou semi-privées).

Bien que JavaScript ne possède pas de mécanisme pour les propriétés strictement privées, l'usage de l'underscore est une convention pour indiquer que ces propriétés ne doivent pas être directement modifiées depuis l'extérieur.

2. Getter pour la propriété name

```
// Getter pour name
    get name() {
        return this._name.toUpperCase();
    }
```

- Le getter est une méthode qui permet de récupérer la valeur de la propriété _name lorsqu'on accède à person3.name.
- Ici, le getter retourne la valeur de **_name** convertie en majuscules avec la méthode **toUpperCase().** Cela signifie que chaque fois qu'on récupère la valeur de name, elle sera affichée en majuscules, peu importe sa forme d'origine.

3. Setter pour la propriété name

```
// Setter pour name
    set name(newName) {
        this._name = newName;
    }
```

- Le setter est une méthode utilisée pour modifier la valeur de la propriété _name. Lorsqu'on essaie de changer la valeur de name via person3.name = 'Emma', le setter est automatiquement appelé et met à jour la propriété _name avec la nouvelle valeur.
- Ici, il n'y a pas de validation particulière pour la modification du nom, la valeur passée est simplement affectée.

4. Getter pour la propriété age

```
    // Getter pour age
```

```
        get age() {
            return this._age;
        }
```

Le getter pour age permet de récupérer la valeur de _age. Ici, il retourne simplement la valeur actuelle de _age.

5. Setter pour la propriété age

```
    // Setter pour age
        set age(newAge) {
            if (newAge > 0) {
                this._age = newAge;
            } else {
                console.log('Âge invalide');
            }
        }
```

- Le setter pour age permet de changer la valeur de _age, mais avec une condition : la nouvelle valeur (newAge) doit être supérieure à 0. Si ce n'est pas le cas, il affiche le message "Âge invalide" dans la console et ne change pas la valeur.
- Cette méthode montre comment les setters peuvent être utilisés pour valider les valeurs avant de les accepter.

6. Utilisation de la classe Person

```
    const person3 = new Person('Lucie', 22);
      console.log(person3.name); // LUCIE (le getter modifie
l'affichage)
     person3.name = 'Emma'; // Utilise le setter pour changer
le nom
    console.log(person3.name); // EMMA


    person3.age = -5; // Affiche "Âge invalide"
```

- **new Person('Lucie', 22)** crée un nouvel objet person3 avec le nom "Lucie" et l'âge 22.
- **console.log(person3.name)** : Lorsque vous essayez de récupérer le nom, le getter est appelé. Comme ce getter transforme le nom en majuscules, la sortie sera "LUCIE" même si le nom d'origine était en minuscules.

- **person3.name = 'Emma'** : Lorsque vous changez le nom en "Emma", le setter est appelé. Le nom est mis à jour à **_name = 'Emma'**. Si vous affichez à nouveau person3.name, le getter renverra "**EMMA**", encore une fois en majuscules grâce à la méthode **toUpperCase().**
- **person3.age = -5** : Lorsque vous essayez de changer l'âge à une valeur négative, le setter pour l'âge détecte que la valeur est invalide et affiche "Âge invalide" dans la console sans changer la valeur actuelle de **_age**.

Points clés sur les getters et setters :

- Les getters sont utilisés pour contrôler l'accès en lecture à une propriété, permettant de transformer ou de formater la valeur avant qu'elle ne soit renvoyée.
- Les setters permettent de contrôler l'accès en écriture, en appliquant potentiellement des règles ou validations avant de modifier la valeur.
- Ces méthodes permettent d'encapsuler les propriétés, offrant une abstraction qui peut être utile pour protéger les données ou pour effectuer des traitements sur les valeurs.

Cela rend le code plus flexible et sécurise les manipulations des propriétés d'un objet en évitant les erreurs ou les incohérences.

Propriétés et méthodes statiques en JavaScript

En JavaScript, les propriétés et méthodes statiques sont des fonctionnalités associées à une classe plutôt qu'aux instances de cette classe. Cela signifie que vous pouvez les appeler directement à partir de la classe sans avoir besoin de créer un objet à partir de celle-ci. Elles sont souvent utilisées pour définir des fonctionnalités qui ne sont pas liées à une instance particulière de la classe, mais qui concernent plutôt la classe elle-même.

Qu'est-ce qu'une méthode statique ?
Une méthode statique est une méthode qui appartient à la classe elle-même et non aux objets créés à partir de cette classe. Pour définir une méthode statique, on utilise le mot-clé **static**. Ces méthodes sont accessibles directement via la classe, sans passer par une instance de la classe.

Exemple de méthode statique :

MathUtils.js

```
    class MathUtils {
          // Méthode statique pour calculer la somme de deux
nombres
        static add(a, b) {
            return a + b;
        }
    }

    // Appel de la méthode statique directement via la classe
    console.log(MathUtils.add(5, 3)); // 8
```

Dans cet exemple, **add** est une **méthode statique**. Vous pouvez appeler cette méthode directement avec **MathUtils.add()** sans avoir besoin de créer une instance de la classe **MathUtils**. Cela est pratique pour des fonctions utilitaires qui ne dépendent pas de l'état d'un **objet particulier**.

Qu'est-ce qu'une propriété statique ?
De la même manière que les méthodes, les propriétés statiques sont associées à la classe et non aux instances. Elles permettent de stocker des informations qui ne changent pas d'une instance à l'autre et sont partagées entre toutes les instances de la classe.

Exemple de propriété statique :

Car.js

```
    class Car {
        // Déclaration d'une propriété statique
        static numberOfWheels = 4;

        constructor(brand) {
            this.brand = brand;
        }

        displayBrand() {
            console.log(`La voiture est une ${this.brand}`);
        }
```

```
}

// Accès à la propriété statique via la classe
console.log(Car.numberOfWheels); // 4

const myCar = new Car('Toyota');
myCar.displayBrand(); // La voiture est une Toyota
```

Dans cet exemple, **numberOfWheels** est une propriété statique. Comme pour les méthodes statiques, on y accède directement via la classe Car avec **Car.numberOfWheels**, et non pas via une instance de la classe.

Pourquoi utiliser des méthodes et propriétés statiques ?
Les méthodes et propriétés statiques sont particulièrement utiles lorsque vous avez besoin de :

- Définir des constantes ou des informations partagées, comme dans l'exemple de la propriété statique **numberOfWheels**.
- Fournir des méthodes utilitaires qui ne nécessitent pas l'accès à une instance (objet) de la classe. Par exemple, la méthode **MathUtils.add()** est une fonction utilitaire qui effectue une opération de base sans nécessiter de propriété ou d'état d'objet.
- Gérer des compteurs globaux ou des informations partagées entre toutes les instances, comme compter combien d'instances d'une classe ont été créées.

Différence entre méthodes normales et méthodes statiques

- **Méthodes normales** : Elles sont définies sans le mot-clé **static** et sont accessibles uniquement via une **instance** (c'est à dire via un objet) de la classe.
- **Méthodes statiques** : Elles sont définies avec le mot-clé **static** et sont accessibles uniquement via la **classe** elle-même.

Voici un exemple qui illustre la différence :

Person.js

```
class Person {
```

```
        constructor(name) {
            this.name = name;
        }

        // Méthode normale
        sayHello() {
            console.log(`Bonjour, je m'appelle ${this.name}`);
        }

        // Méthode statique
        static createRandomPerson() {
                const randomName = ['Alice', 'Bob', 'Charlie']
[Math.floor(Math.random() * 3)];
            return new Person(randomName);
        }
    }

    const person1 = new Person('John');
    person1.sayHello(); // Bonjour, je m'appelle John

    const randomPerson = Person.createRandomPerson();
      randomPerson.sayHello(); // Bonjour, je m'appelle Alice
(ou Bob, ou Charlie)
```

Dans cet exemple :

- La méthode **sayHello** est une méthode normale. Elle nécessite une instance de la classe Person pour être appelée, car elle fait référence à une propriété d'instance (**this.name**).
- La méthode **createRandomPerson** est une méthode statique. Elle n'a pas besoin d'instance et peut être appelée directement via la classe Person. Elle crée une instance de Person sans se baser sur une instance existante.

Limitations des méthodes statiques

Une méthode statique ne peut pas accéder aux propriétés d'instance de la classe (comme this.name) car elle n'est pas appelée sur une instance spécifique.
Par ailleurs, les méthodes normales ne peuvent pas appeler directement des méthodes statiques. Si vous voulez le faire, vous devrez utiliser la syntaxe **ClassName.methodName()**.

Exemple de limitation :

Animal.js

```
    class Animal {
        constructor(name) {
            this.name = name;
        }

        // Méthode statique
        static getClassName() {
            return 'Animal';
        }

        // Méthode normale
        showInfo() {
             // Impossible d'appeler directement getClassName() car elle est statique
              console.log(`Ceci est un ${Animal.getClassName()} nommé ${this.name}`);
        }
    }

    const dog = new Animal('Buddy');
    dog.showInfo(); // Ceci est un Animal nommé Buddy
```

Dans cet exemple, **getClassName** est une méthode statique. Pour l'appeler depuis une méthode d'instance (comme **showInfo**), il faut la référencer via le nom de la classe **Animal.getClassName()**, car **this.getClassName()** ne fonctionnerait pas (méthode statique inaccessible depuis une instance).

En conclusion, les méthodes et propriétés statiques sont des outils pratiques pour structurer votre code de manière claire et efficace.

Elles permettent de regrouper les fonctions utilitaires ou les données partagées à travers une classe, tout en évitant de devoir créer des instances inutiles. Cependant, il est important de bien comprendre la différence entre une méthode statique et une méthode d'instance pour les utiliser correctement.

Conclusion

Dans ce chapitre, nous avons exploré en détail la création et l'utilisation des classes et des objets en JavaScript. Vous avez appris à définir des classes, à créer des objets et à utiliser des constructeurs pour initialiser les propriétés d'un objet dès sa création. Les méthodes, que ce soit pour manipuler ou afficher les données, permettent d'ajouter du comportement à vos objets et d'organiser votre code de manière plus lisible et efficace.

Nous avons également abordé deux mécanismes essentiels : les accesseurs (getters) et les mutateurs (setters), qui offrent un contrôle plus fin sur la manière dont les propriétés des objets sont lues et modifiées. Ces outils vous permettent de valider les données ou de modifier leur présentation sans changer la manière dont les objets sont utilisés. En plus de cela, nous avons couvert les propriétés et méthodes statiques, qui se rapportent directement à la classe elle-même, plutôt qu'à ses instances. Celles-ci sont très utiles pour regrouper des fonctionnalités générales ou des constantes.

La maîtrise de ces concepts est un pas crucial vers une meilleure organisation de vos applications JavaScript. Les classes et objets sont des piliers fondamentaux dans la programmation orientée objet (POO), qui favorise une approche modulaire, claire et extensible du code. En structurant vos programmes de cette manière, vous faciliterez non seulement la maintenance, mais aussi la réutilisation du code.

Dans le chapitre suivant, nous allons approfondir l'un des concepts clés de la POO : l'héritage. L'héritage vous permettra de créer des relations entre les classes, de réutiliser du code existant et de construire des systèmes plus sophistiqués où les classes peuvent être étendues et spécialisées. Vous verrez comment, grâce à l'héritage, vous pourrez créer des classes dérivées qui héritent des caractéristiques et comportements d'autres classes, tout en ajoutant ou modifiant leurs propres fonctionnalités.

Ainsi, l'héritage vous offrira un cadre puissant pour organiser votre code de manière hiérarchique, flexible et évolutive. À bientôt pour

découvrir comment ce concept enrichit vos compétences en programmation orientée objet !

Chapitre 10

Principe d'encapsulation en JavaScript

L'encapsulation est l'un des principes fondamentaux de la programmation orientée objet (POO). Elle consiste à regrouper les données (propriétés) et les méthodes (comportements) qui manipulent ces données au sein d'un même objet ou d'une même classe. En JavaScript, ce concept permet de protéger l'intégrité des données en contrôlant l'accès et la modification des propriétés internes de l'objet. En d'autres termes, cela signifie que certaines propriétés ou méthodes sont privées et ne peuvent pas être directement modifiées ou accessibles depuis l'extérieur de l'objet.

L'objectif principal de l'encapsulation est d'empêcher des modifications ou utilisations non souhaitées des données, en fournissant des méthodes sécurisées pour y accéder ou les modifier. Ce mécanisme améliore la sécurité, la maintenance et la flexibilité du code.

Dans ce chapitre, nous allons explorer en détail le principe de l'encapsulation en JavaScript, en abordant notamment :

- La différence entre propriétés publiques et privées.
- Comment utiliser des getters et setters pour contrôler l'accès aux données.
- L'utilisation des symboles et des conventions pour encapsuler des données.

Propriétés publiques et privées

En JavaScript, par défaut, toutes les propriétés et méthodes définies dans une classe sont publiques, c'est-à-dire qu'elles peuvent être accédées et modifiées directement depuis l'extérieur de l'objet. Toutefois, pour suivre le principe de l'encapsulation, il est essentiel de restreindre l'accès à certaines propriétés ou méthodes.

Traditionnellement, JavaScript ne disposait pas de mécanismes intégrés pour créer des propriétés véritablement privées, contrairement à d'autres langages de programmation. Cependant, en utilisant certaines conventions, telles que l'ajout d'un underscore (_) avant le nom de la propriété, il est possible d'indiquer qu'une propriété est "privée" et ne doit pas être modifiée directement.

Voici un exemple basique d'encapsulation avec des conventions :

```
class Person {
    constructor(firstName, lastName) {
        this._firstName = firstName; // Propriété privée
        this._lastName = lastName;
    }

    // Getter pour firstName
    get firstName() {
        return this._firstName;
    }

    // Setter pour firstName
    set firstName(newFirstName) {
        this._firstName = newFirstName;
    }

    // Getter pour lastName
    get lastName() {
        return this._lastName;
    }

    // Setter pour lastName
    set lastName(newLastName) {
```

```
        this._lastName = newLastName;
    }
}

const person1 = new Person('John', 'Doe');
console.log(person1.firstName); // John
person1.firstName = 'Jane'; // Modifier le prénom
console.log(person1.firstName); // Jane
```

Dans cet exemple, les propriétés **_firstName** et **_lastName** sont considérées comme privées grâce à la convention de nommage avec l'underscore. Pour interagir avec ces propriétés, des accesseurs (get) et des mutateurs (set) sont utilisés, ce qui permet de contrôler l'accès et la modification de ces valeurs.

Propriétés privées avec les champs privés

Depuis ECMAScript 2020, JavaScript permet de créer des propriétés privées en utilisant le symbole #. Ces propriétés ne sont accessibles qu'à l'intérieur de la classe et ne peuvent pas être modifiées directement depuis l'extérieur.

Voici un exemple utilisant cette nouvelle syntaxe :

```
class Car {
    #make;
    #model;

    constructor(make, model) {
        this.#make = make; // Propriétés privées
        this.#model = model;
    }

    // Méthode pour obtenir les informations de la voiture
    getDetails() {
        return `${this.#make} ${this.#model}`;
    }

    // Méthode pour modifier le modèle de la voiture
    setModel(newModel) {
```

```
            this.#model = newModel;
        }
    }

    const car1 = new Car('Toyota', 'Corolla');
    console.log(car1.getDetails()); // Toyota Corolla
    car1.setModel('Camry'); // Changer le modèle
    console.log(car1.getDetails()); // Toyota Camry
```

Ici, les propriétés **#make** et **#model** sont véritablement privées et ne peuvent pas être accédées ou modifiées directement depuis l'extérieur de la classe Car.

Utilisation des getters et setters pour l'encapsulation

L'encapsulation en JavaScript peut également être gérée via des getters et des setters. Les getters permettent d'accéder aux valeurs des propriétés privées, tandis que les setters permettent de les modifier de manière contrôlée.

```
    class BankAccount {
        #balance;

        constructor(balance) {
            this.#balance = balance;
        }

        // Getter pour récupérer le solde
        get balance() {
            return this.#balance;
        }

            // Setter pour mettre à jour le solde de manière
contrôlée
        set balance(newBalance) {
            if (newBalance >= 0) {
                this.#balance = newBalance;
            } else {
```

```
                    console.log('Le solde ne peut pas être
négatif');
            }
        }
    }

    const account1 = new BankAccount(100);
    console.log(account1.balance); // 100
    account1.balance = 200; // Mettre à jour le solde
    console.log(account1.balance); // 200
     account1.balance = -50; // Erreur, le solde ne peut pas
être négatif
```

Avantages de l'encapsulation

L'encapsulation en JavaScript offre plusieurs avantages :

- **Protection des données** : En rendant certaines propriétés privées, vous pouvez éviter que des valeurs internes soient accidentellement modifiées ou corrompues par des parties externes du code.
- **Modularité** : L'encapsulation permet de séparer les responsabilités dans un programme, rendant le code plus facile à maintenir et à déboguer.
- **Contrôle des accès** : En utilisant des getters et des setters, vous pouvez définir des règles spécifiques pour accéder et modifier les propriétés d'un objet, ce qui garantit une gestion plus sûre des données.

Conclusion

L'encapsulation est un concept clé en programmation orientée objet. En JavaScript, elle permet de protéger les propriétés internes d'une classe et de les rendre accessibles de manière contrôlée grâce à l'utilisation de getters et de setters. Grâce à l'introduction des propriétés privées avec le symbole #, JavaScript offre désormais un moyen plus robuste de mettre en œuvre ce principe. L'encapsulation joue un rôle essentiel dans la création de code modulaire, sûr et maintenable.

Dans le chapitre suivant, nous aborderons les notions d'héritage, une autre caractéristique essentielle de la programmation orientée objet qui permet de créer des relations entre les classes et de réutiliser du code.

Chapitre 11

L'héritage en JavaScript

L'un des concepts les plus puissants de la programmation orientée objet est l'héritage. L'héritage permet de créer des relations entre différentes classes en permettant à une classe d'hériter des propriétés et des méthodes d'une autre. En d'autres termes, l'héritage vous permet de créer des classes dérivées à partir de classes existantes, facilitant ainsi la réutilisation du code et la création de structures plus modulaires et évolutives.

Dans le cadre de JavaScript, qui repose sur un modèle orienté prototype, l'héritage prend une forme particulière mais extrêmement flexible. Avec l'introduction de la syntaxe class en ES6, JavaScript a simplifié la manière dont les développeurs peuvent implémenter l'héritage, rendant le langage plus accessible à ceux qui viennent d'autres langages orientés objet.

L'héritage en JavaScript est utilisé lorsque vous souhaitez qu'une nouvelle classe étende ou améliore les fonctionnalités d'une classe existante. Cela vous permet d'éviter de dupliquer le code et de partager les comportements communs entre plusieurs classes. Par exemple, imaginez une classe de base Personne qui possède des propriétés et méthodes communes à toutes les personnes. À partir de cette classe, vous pourriez créer des classes dérivées comme Étudiant ou Professeur, qui hériteraient des fonctionnalités de Personne tout en ajoutant leurs propres spécificités.

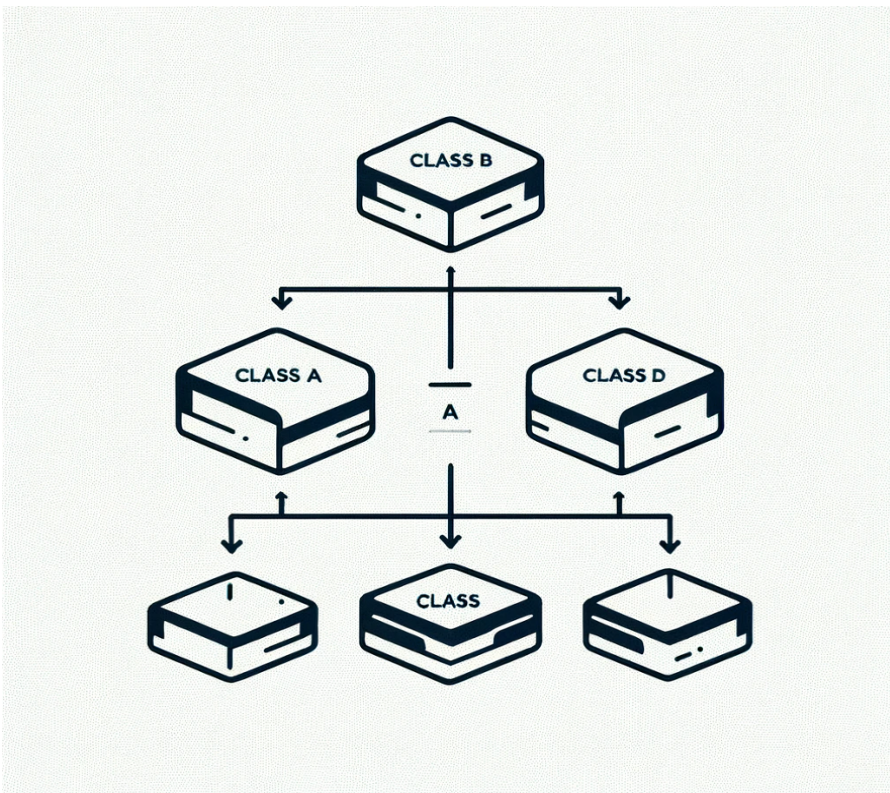
Dans ce chapitre, nous allons explorer comment implémenter l'héritage en JavaScript en utilisant les classes. Vous découvrirez comment créer des classes parents et enfants, comment utiliser le mot-clé extends, et comment réutiliser ou étendre les méthodes et propriétés de la classe parent à l'intérieur des classes dérivées. Grâce à l'héritage, vous pourrez structurer vos applications de manière plus propre et plus efficace, tout en maintenant un code DRY (Don't Repeat Yourself).

Prêt à découvrir comment l'héritage peut transformer vos classes et objets JavaScript en structures puissantes et réutilisables ? Plongeons dans ce concept essentiel de la programmation orientée objet.

Définition de l'héritage en programmation orientée objet

L'héritage est l'un des principes fondamentaux de la programmation orientée objet (POO). Il permet à une classe d'hériter des propriétés et des méthodes d'une autre classe. La classe qui hérite est appelée classe dérivée ou classe enfant, tandis que la classe d'origine est appelée classe parente ou classe de base. Ce mécanisme permet de réutiliser et d'étendre le code existant, tout en facilitant l'organisation et la maintenance des programmes.

En termes simples, l'héritage permet de définir une relation "est-un" entre deux classes. Par exemple, une **classe Animal** peut servir de classe **parente** pour des classes plus spécifiques comme **Dog** ou **Cat**. Ces classes enfants héritent des propriétés communes de la classe Animal (par exemple, breathe (respirer), eat (manger) et peuvent également définir leurs propres méthodes ou propriétés spécifiques (par exemple, un Dog peut avoir une méthode bark (aboie)).



Pourquoi utiliser l'héritage ?

L'un des principaux avantages de l'héritage est la réutilisation du code. Plutôt que de redéfinir les mêmes propriétés et méthodes dans plusieurs classes, vous pouvez les centraliser dans une classe parente et les rendre disponibles pour toutes les classes enfants. Cela permet de réduire la duplication de code et de faciliter les modifications. Si une fonctionnalité doit être changée ou améliorée, il suffit de la modifier dans la classe parente, et toutes les classes dérivées bénéficieront automatiquement de cette mise à jour.

L'héritage favorise également une meilleure organisation et une plus grande modularité du code. Il permet de définir des relations claires entre les objets, ce qui rend le code plus facile à comprendre et à maintenir.

L'héritage en JavaScript est implémenté à l'aide du mot-clé **extends**, qui permet à une classe enfant d'étendre une classe parente en héritant de ses membres publics et protégés. Cela signifie que la classe enfant peut accéder aux propriétés et méthodes définies dans la classe parente comme si elles faisaient partie de sa propre définition. Grâce à cet héritage, il devient possible de réutiliser le code de la classe parente sans avoir à le réécrire, tout en permettant à la classe enfant d'ajouter ou de modifier des comportements spécifiques.

Exemple d'héritage
Prenons un exemple simple d'une classe **parente Person**, qui définit des **propriétés** et **méthodes** de base communes à toutes les personnes, et une classe enfant **Student** qui hérite de **Person** tout en ajoutant des fonctionnalités spécifiques :

```
    // Classe parente
    class Person {
        constructor(name, age) {
            this.name = name;
            this.age = age;
        }

        introduceYourself() {
             console.log(`Bonjour, je m'appelle ${this.name} et
j'ai ${this.age} ans.`);
        }
    }

    // Classe enfant
```

```
    class Student extends Person {
        constructor(name, age, subject) {
            super(name, age); // Appelle le constructeur de la
classe parente
            this.subject = subject;
        }

        study() {
                                console.log(`${this.name}  étudie
${this.subject}.`);
        }
    }

    const student1 = new Student('Alice', 21, 'Informatique');
      student1.introduceYourself(); // Bonjour, je m'appelle
Alice et j'ai 21 ans.
       student1.study();                        // Alice étudie
Informatique.
```

Explication :

Définition de la classe Person :

Dans ce code nous avons créé une classe appelée Person qui servira de modèle pour créer des objets représentant des personnes.
Le constructeur qui est une méthode spéciale automatiquement appelée lors de la création d'une nouvelle instance de la classe prend ici deux arguments, name et age, qui correspondent aux propriétés d'une personne.

- **this.name = name** : Le mot-clé this fait référence à l'instance actuelle de la classe. Cela signifie que nous attribuons la valeur name passée lors de la création de l'objet à la propriété name de l'instance.
- **this.age = age** : De la même manière, age est assigné à la propriété age de l'instance.

dans la classe Person nous avons défini la méthode **introduceYourself()**. Lorsqu'elle est appelée, elle affiche un message dans la console qui inclut le nom et l'âge de la personne (utilisant this.name et this.age).

Définition de la classe enfant : Student

- **class Student extends Person** : Cette ligne définit une classe appelée **Student** qui hérite de la classe Person. Le mot-clé **extends** permet à la classe enfant **Student** de hériter des propriétés et des méthodes définies dans la classe parente Person.
- **constructor(name, age, subject)** : Le constructeur de la classe **Student** prend trois paramètres : **name**, **age**, et un nouveau paramètre **subject** (matière d'étude). Comme Student hérite de Person, il doit initialiser les propriétés name et age définies dans la classe parente.
    - **super(name, age)** : Le mot-clé **super** est utilisé pour appeler le constructeur de la classe parente (Person). Cela permet à la classe enfant Student de transmettre les paramètres name et age au constructeur de Person, qui se chargera de les initialiser.
    - **this.subject = subject** : Ensuite, Student ajoute une propriété supplémentaire subject (matière d'étude) qui est propre à cette classe enfant.
- **study()** : La méthode **study()** est spécifique à la classe Student et affiche un message indiquant le nom de l'étudiant et la matière qu'il étudie (utilisant this.name et **this.subject)**.

Création d'une instance de Student

- **const student1 = new Student('Alice', 21, 'Informatique')** : Cette ligne crée une instance de la classe Student en passant les arguments **name**, **age** et **subject**. Ici, un objet **student1** est créé, représentant une étudiante nommée Alice, âgée de 21 ans, qui étudie l'informatique.
- Le constructeur de Student est appelé, et il utilise le mot-clé **super()** pour transmettre les paramètres name et age au constructeur de Person. La propriété subject est également initialisée pour cette instance.

Utilisation des méthodes de l'objet `student1`

- **student1.introduceYourself()** : Cette méthode est héritée de la classe parente Person. Lorsqu'elle est appelée, elle affiche le message : "Bonjour, je m'appelle Alice et j'ai 21 ans.".

- **student1.study()** : Cette méthode est spécifique à la classe enfant Student. Elle affiche le message : "Alice étudie Informatique.".

Résumé :

- **Héritage** : La classe Student hérite de Person, ce qui permet de réutiliser le code et d'éviter de redéfinir les propriétés name et age, ainsi que la méthode introduceYourself().
- **super()** : Le mot-clé super() permet à la classe enfant d'appeler le constructeur de la classe parente pour initialiser les propriétés héritées.
- **Méthodes spécifiques** : La classe Student peut ajouter ses propres méthodes comme study(), tout en ayant accès aux méthodes héritées comme introduceYourself().

Lorsque vous exécutez ce code, les deux méthodes **introduceYourself()** et **study()** sont appelées. Voici ce qui est affiché dans la console :

```
Bonjour, je m'appelle Alice et j'ai 21 ans.
Alice étudie Informatique.
```

Ce comportement montre que :

- La classe Student hérite correctement de la classe Person et peut accéder aux méthodes et propriétés de cette dernière.
- La méthode spécifique study() définie dans Student fonctionne également comme attendu, en utilisant les valeurs de l'instance student1.

Redéfinition des méthodes de la classe mère

En programmation orientée objet, il est possible pour une classe enfant de redéfinir ou surcharger une méthode qui a été définie dans sa classe parente. Ce processus s'appelle la redéfinition de méthode (ou override en anglais). La redéfinition permet à la classe enfant de fournir une nouvelle implémentation de la méthode héritée, adaptée à ses besoins spécifiques.

Pourquoi redéfinir une méthode ?
La redéfinition d'une méthode est utile lorsque la classe enfant a

besoin de modifier le comportement d'une méthode héritée tout en conservant la structure générale de la classe parente. Cela permet de personnaliser les fonctionnalités de la classe enfant sans avoir à réécrire entièrement le code de la classe parente.

Exemple pratique de redéfinition de méthode
Prenons l'exemple précédent avec une classe Person et une classe Student. Supposons que la classe Person ait une méthode **introduceYourself()**, mais que la classe Student souhaite modifier cette méthode pour ajouter une information spécifique à un étudiant, comme sa matière d'étude.

Voici comment **redéfinir** cette méthode dans la classe **student**:

```
// Classe parente
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    introduceYourself() {
            console.log(`Bonjour, je m'appelle ${this.name} et
j'ai ${this.age} ans.`);
    }
}

    // Classe enfant
    class Student extends Person {
        constructor(name, age, subject) {
             super(name, age); // Appelle le constructeur de la
classe parente
            this.subject = subject;
        }

        // Redéfinition de la méthode introduceYourself()
        introduceYourself() {
              console.log(`Bonjour, je m'appelle ${this.name},
j'ai ${this.age} ans, et j'étudie ${this.subject}.`);
        }
```

```
        study() {
                                console.log(`${this.name}  étudie
${this.subject}.`);
        }
    }

    // Création d'une instance de Student
    const student1 = new Student('Alice', 21, 'Informatique');
      student1.introduceYourself(); // Bonjour, je m'appelle
Alice, j'ai 21 ans, et j'étudie Informatique.              //
Alice étudie Informatique.
```

Explications:

- **introduceYourself()** dans la classe Person : Cette méthode affiche un message standard pour une personne, incluant son nom et son âge.
- Redéfinition de **introduceYourself()** dans **Student** : Dans la classe enfant Student, la méthode est redéfinie pour ajouter une information supplémentaire, à savoir la matière étudiée par l'étudiant (subject).
- **Appel de la méthode** : Lorsque **student1.introduceYourself()** est appelée, c'est la version redéfinie dans la classe **Student** qui est exécutée, et non celle de Person.

Ainsi, même si **introduceYourself()** est défini dans la classe parente, la classe enfant peut fournir sa propre version de cette méthode pour l'adapter à ses besoins spécifiques.

Lorsque vous exécutez ce code, vous obtiendrez le résultat :

```
Bonjour, je m'appelle Alice, j'ai 21 ans, et j'étudie Informatique.
```

Appel de la méthode redéfinie et de la méthode parente
Dans certains cas, il peut être nécessaire d'utiliser à la fois la méthode redéfinie et la méthode de la classe parente. Cela est possible grâce au mot-clé super qui permet d'accéder à la méthode d'origine dans la classe parente.

Voyons comment faire cela :

Modifications de la méthode introduceYourself() dans la classe Student :

```
    // Classe enfant
    class Student extends Person {
        constructor(name, age, subject) {
            super(name, age); // Appelle le constructeur de la classe parente
            this.subject = subject;
        }

        // Redéfinition de la méthode introduceYourself()
        introduceYourself() {
            // Appelle la méthode parente avec super
            super.introduceYourself();
                    console.log(`Je suis étudiant et j'étudie ${this.subject}.`);
        }
    }
```

Explications :

super.introduceYourself() : Le mot-clé super permet d'appeler la méthode introduceYourself() définie dans la classe parente Person. Cela signifie que la version originale de la méthode est exécutée avant d'ajouter la nouvelle logique propre à la classe enfant Student.

Appel combiné de la méthode introduceYourself() :

```
    // Création d'une instance de Student
    const student2 = new Student('Bob', 22, 'Mathématiques');
    student2.introduceYourself(); // Bonjour, je m'appelle Bob et j'ai 22 ans.
    // Je suis étudiant et j'étudie Mathématiques.
```

Lorsque **student2.introduceYourself()** est appelé, il commence par exécuter le comportement de la méthode parente (super.introduceYourself()) et ensuite ajoute une nouvelle phrase spécifique à l'étudiant.

Résultat :

```
Bonjour, je m'appelle Bob et j'ai 22 ans.
Je suis étudiant et j'étudie Mathématiques.
```

La redéfinition de méthodes dans les classes enfants est un aspect clé de la programmation orientée objet. Elle permet d'adapter et d'étendre le comportement des classes parentes sans avoir à réécrire entièrement leur code. Grâce à la redéfinition, la classe enfant peut personnaliser les méthodes héritées tout en conservant la structure de base fournie par la classe parente. De plus, avec le mot-clé super, il est possible d'appeler la méthode parente tout en ajoutant des fonctionnalités supplémentaires dans la classe enfant.

Héritage multiniveau en JavaScript

L'héritage multiniveau est un concept de la programmation orientée objet où une classe dérive d'une autre classe, qui elle-même dérive d'une troisième classe. En d'autres termes, dans un héritage multiniveau, une chaîne de classes est créée, où chaque classe hérite des propriétés et des méthodes de sa classe parente, et peut elle-même être la classe parente d'une autre classe.

En JavaScript, bien que l'héritage ne soit pas profondément imbriqué comme dans certains langages orientés objet purs, il est tout à fait possible d'implémenter un héritage multiniveau grâce à la syntaxe des classes et au mot-clé extends. Cela permet de réutiliser le code sur plusieurs niveaux de classes et de définir des comportements spécifiques à chaque niveau d'héritage.

Exemple pratique d'héritage multiniveau
Voyons un exemple d'héritage multiniveau avec trois classes : Person, Student, et GraduateStudent. Dans cet exemple, Student hérite de Person, et GraduateStudent hérite de Student.

```
// Première classe : Person
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    introduceYourself() {
         console.log(`Bonjour, je m'appelle ${this.name} et
```

```
j'ai ${this.age} ans.`);
        }
    }

    // Deuxième classe : Student, qui hérite de Person
    class Student extends Person {
        constructor(name, age, subject) {
            super(name, age); // Appelle le constructeur de la
classe parente (Person)
            this.subject = subject;
        }

        study() {
                                console.log(`${this.name}  étudie
${this.subject}.`);
        }
    }

      // Troisième classe : GraduateStudent, qui hérite de
Student
    class GraduateStudent extends Student {
        constructor(name, age, subject, thesisTopic) {
                    super(name, age, subject); // Appelle le
constructeur de la classe parente (Student)
            this.thesisTopic = thesisTopic;
        }

        research() {
             console.log(`${this.name} fait des recherches sur
${this.thesisTopic}.`);
        }
    }

    // Création d'une instance de GraduateStudent
      const gradStudent1 = new GraduateStudent('Alice', 25,
'Informatique', 'Intelligence Artificielle');

    // Appel des méthodes héritées et propres
         gradStudent1.introduceYourself();     // Bonjour, je
```

```
m'appelle Alice et j'ai 25 ans.
      gradStudent1.study();                    // Alice étudie
Informatique.
      gradStudent1.research();                 // Alice fait des
recherches sur Intelligence Artificielle.
```

Explications :

- **Classe Person** : Cette classe définit les propriétés name et age ainsi qu'une méthode introduceYourself(), qui affiche les informations de base d'une personne.
- **Classe Student** : La classe Student hérite de Person et ajoute une nouvelle propriété subject (matière) et une méthode study(), spécifique aux étudiants.
- **Classe GraduateStudent** : Cette classe hérite à son tour de Student et ajoute une propriété thesisTopic (sujet de thèse), ainsi qu'une méthode research() pour représenter une recherche effectuée par l'étudiant diplômé.

Appels des méthodes :

- Lorsque l'instance gradStudent1 est créée à partir de la classe GraduateStudent, elle a accès aux méthodes héritées des deux classes parents (Person et Student), ainsi qu'à sa propre méthode research().
- introduceYourself() : Appelle la méthode définie dans Person via l'héritage multiniveau.
- study() : Appelle la méthode définie dans Student.
- research() : Appelle la méthode définie dans GraduateStudent.

Avantages de l'héritage multiniveau

- Réutilisation maximale du code : Avec l'héritage multiniveau, chaque classe enfant hérite des fonctionnalités de ses ancêtres, évitant ainsi la duplication du code.
- Modularité : Le comportement peut être modulé à chaque niveau d'héritage. Par exemple, GraduateStudent peut étendre le comportement de Student tout en conservant les fonctionnalités de Person.
- Spécificité : À chaque niveau d'héritage, des propriétés et des méthodes spécifiques à la classe peuvent être ajoutées sans affecter les classes parents.

Attention aux risques de l'héritage multiniveau

- Complexité accrue : Bien que l'héritage multiniveau favorise la réutilisation du code, il peut également ajouter de la complexité si les relations d'héritage deviennent trop profondes.
- Dépendance excessive : Une chaîne d'héritage trop longue peut conduire à une forte dépendance entre les classes, rendant les modifications plus difficiles à gérer si une classe parent doit être modifiée.

En résumé, l'héritage multiniveau permet de structurer des chaînes de classes qui héritent les unes des autres en créant des relations hiérarchiques. Ce type d'héritage est utile pour organiser le code et réutiliser des fonctionnalités sur plusieurs niveaux, tout en ajoutant des comportements spécifiques à chaque étape. Cependant, il convient d'utiliser l'héritage multiniveau avec précaution, afin d'éviter une complexité excessive et des dépendances trop profondes dans le code.

Conclusion

L'héritage est un concept fondamental de la programmation orientée objet, et JavaScript l'implémente de manière efficace avec la syntaxe moderne introduite par ES6. Grâce à l'héritage, vous pouvez réutiliser et étendre des fonctionnalités, ce qui vous permet de structurer vos applications de manière modulaire et maintenable. En créant des classes parents et classes enfants, vous pouvez éviter la duplication de code tout en personnalisant le comportement des classes dérivées.

Dans ce chapitre, nous avons vu :

- Comment implémenter l'héritage avec le mot-clé extends.
- Comment redéfinir des méthodes pour adapter le comportement des classes enfants.
- Le concept d'héritage multiniveau, qui permet d'étendre des fonctionnalités sur plusieurs niveaux de classes.

L'héritage est un outil puissant qui simplifie la gestion des grandes applications en regroupant des comportements communs dans des classes parente, tout en permettant aux classes enfants de se spécialiser. Toutefois, il est important d'utiliser cet outil avec précaution pour éviter une complexité excessive et des relations d'héritage trop profondes.

En maîtrisant l'héritage, vous serez en mesure de créer des

architectures de code plus propres, plus réutilisables et plus efficaces, ce qui est essentiel dans tout projet JavaScript de grande envergure. Dans les prochains chapitres, nous explorerons d'autres concepts avancés de JavaScript pour continuer à améliorer vos compétences dans ce langage incontournable.

Chapitre 12

Les Prototypes en JavaScript

JavaScript est souvent décrit comme un langage orienté objet, mais il se distingue des autres langages orientés objet comme Java ou C++ en raison de son modèle basé sur les **prototypes** plutôt que sur les **classes**. Comprendre comment fonctionnent les prototypes est essentiel pour maîtriser les concepts sous-jacents de JavaScript. Dans ce chapitre, nous allons explorer ce qu'est un prototype, comment l'héritage se fait à travers la chaîne des prototypes, et en quoi JavaScript, malgré sa syntaxe moderne de classes, reste fondamentalement orienté prototype.

Qu'est-ce qu'un prototype ? : Le modèle basé sur les prototypes en JavaScript

En JavaScript, chaque objet possède un lien caché vers un autre objet appelé prototype. Ce prototype est une source à laquelle l'objet peut accéder pour obtenir des propriétés ou des méthodes. Si un objet n'a pas une méthode ou une propriété qui est demandée, JavaScript va chercher cette méthode ou propriété dans son prototype. Cette recherche se poursuit dans la chaîne des prototypes jusqu'à atteindre le prototype final, souvent Object.prototype, qui est la racine de tous les objets en JavaScript.

Par exemple :

Créons un script person.js que nous lions à une page index.html

```
    const person = {
        name: 'Alice',
        age: 30,
        introduce() {
            console.log(`Bonjour, je m'appelle ${this.name} et
j'ai ${this.age} ans.`);
        }
    };

    // L'objet 'person' a un prototype par défaut
       console.log(person.__proto__  ===  Object.prototype);  //
true
```

Explication :

- **Création de l'objet person :**

La première partie de ce code crée un objet simple appelé **person** avec trois propriétés :

- **name** : Valeur de 'Alice'.
- **age** : Valeur de 30.
- **introduce()** : Une méthode qui affiche un message dans la console avec le nom et l'âge de la personne.

Cet objet (person) est une représentation simple d'une personne avec

un nom, un âge et une fonction qui lui permet de se présenter. Rien dans cet objet ne semble faire référence à un **prototype** pour l'instant. Cependant, en JavaScript, tous les objets (y compris l'objet person) ont un lien implicite vers un autre **objet** appelé **prototype**.

- **Le prototype de person :**

Chaque objet JavaScript possède un lien caché vers un autre objet qui s'appelle son prototype. Ce lien prototype est stocké dans la propriété _proto_ de l'objet. Ce n'est pas une méthode que vous définissez vous-même, elle est créée automatiquement par JavaScript.

Dans le cas de l'objet person, puisque nous avons créé cet objet de manière littérale (avec des accolades {}), son prototype est automatiquement lié à Object.prototype, qui est le prototype de base de tous les objets créés littéralement en JavaScript.

Regardons cette ligne du code :

```
console.log(person.__proto__ === Object.prototype); // true
```

- **person._proto_** : C'est la référence au prototype de l'objet **person**.
- **Object.prototype** : C'est le prototype de base de tous les objets JavaScript créés littéralement.

Lorsque nous comparons **person._proto_** avec **Object.prototype**, le résultat est true, ce qui signifie que l'objet person hérite de Object.prototype.

Ajoutons maintenant les deux lignes de code suivantes et exécutons le script person.js :

```
console.log(person.__proto__);
console.log(Object.prototype);
```

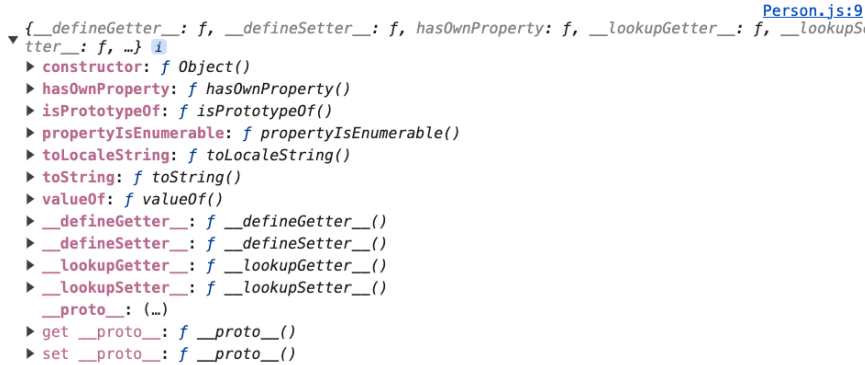
Exécution de la première ligne :

Lorsque la première ligne est exécutée, JavaScript affiche le prototype de l'objet person. En JavaScript, tous les objets possèdent une propriété **cachée** appelée **_proto_**. Cette propriété pointe vers le prototype de l'objet, qui est l'objet dont il hérite les propriétés et méthodes.

Dans ce cas précis, person a été créé comme un objet littéral (avec des accolades **{}**). Par défaut, les objets créés de cette manière héritent de **Object.prototype**, le prototype de base pour tous les objets créés

littéralement en JavaScript.

Le résultat de cette ligne sera donc quelque chose qui ressemble à ceci dans la console :



Ce résultat montre le contenu du prototype de l'objet **person**, qui est en fait **Object.prototype**. Ce prototype contient plusieurs méthodes natives de JavaScript telles que :

- **constructor** : Spécifie la fonction qui crée une instance de l'objet.
- **hasOwnProperty** : Vérifie si une propriété appartient à l'objet lui-même (et non à son prototype).
- **isPrototypeOf** : Vérifie si un objet se trouve dans la chaîne de prototypes de l'objet spécifié.
- **toString** : Convertit l'objet en chaîne de caractères.

Ce sont des méthodes disponibles pour tous les objets créés via un objet littéral en JavaScript.

Exécution de la deuxème ligne :

La deuxième ligne du code précédent affiche directement le prototype de Object lui-même, qui est le prototype de base pour tous les objets créés en JavaScript. Lorsque vous utilisez la syntaxe d'objet littéral {}, l'objet que vous créez a **Object.prototype** comme prototype.

Le résultat dans la console sera identique à celui de la première ligne :

Ce résultat montre que Object.prototype contient les méthodes fondamentales que tous les objets JavaScript héritent, y compris :

Constructor, hasOwnProperty, toString, valueOf etc.
Ces méthodes sont communes à tous les objets créés dans JavaScript, car tous les objets littéraux et la plupart des objets en général héritent

directement ou indirectement de **Object.prototype**.

Prototype et héritage : Comment les objets héritent les uns des autres via la chaîne des prototypes

L'héritage par prototypes est un mécanisme où les objets peuvent hériter d'autres objets. Contrairement aux langages basés sur des classes, où vous créez des instances d'une classe, en JavaScript, l'héritage se fait par la chaîne des prototypes. Chaque objet a un prototype, et ce prototype peut à son tour avoir un prototype, formant une chaîne.

Prenons un exemple simple :

myanimal.js

```
const animal = {
    breathe() {
        console.log('Je respire.');
    }
};

const dog = {
    bark() {
        console.log('Wouf!');
    }
};

// On définit le prototype de 'dog' comme étant 'animal'
Object.setPrototypeOf(dog, animal);

dog.bark();    // Wouf!
dog.breathe(); // Je respire.
```

Explication:

Dans ce code, nous avons créé littéralement un objet simple, appelé **animal** qui possède une méthode **breathe()**. Lorsque cette méthode est appelée, elle affiche le message "Je respire.". Cela simule un comportement de base pour un animal : respirer.

Puis, nous avons créé littéralement un deuxième objet **dog** qui possède une méthode appelée **bark()** qui affiche le message "Wouf!" lorsqu'elle est appelée. Cela représente le comportement spécifique d'un chien : aboyer.

Pour l'instant, dog ne sait faire qu'aboyer et animal ne sait que respirer. Ils n'ont aucun lien entre eux. Si nous essayions d'appeler la méthode breathe() sur dog, cela ne fonctionnerait pas encore, car l'objet dog ne connaît pas cette méthode.

Ensuite, nous avons ajouté la ligne de code :

```
Object.setPrototypeOf(dog, animal);
```

Cette ligne est cruciale. Elle utilise la méthode **Object.setPrototypeOf()** pour définir le prototype de l'objet **dog** comme étant l'objet **animal**.

- En JavaScript, un **prototype** est un **lien caché** qu'un objet peut avoir vers un autre objet. Lorsqu'une propriété ou une méthode n'est pas trouvée sur l'objet lui-même, JavaScript remonte la chaîne des prototypes pour la chercher.
- En définissant animal comme **prototype** de dog, nous permettons à dog d'accéder aux méthodes définies dans animal, en plus de ses propres méthodes.

Ainsi, maintenant que **dog** a **animal** comme prototype, il pourra accéder à la méthode **breathe()** de animal, même si cette méthode n'est pas directement définie sur dog.

Par conséquent, nous pouvons appeler la méthode **breathe()** sur l'objet **dog**.

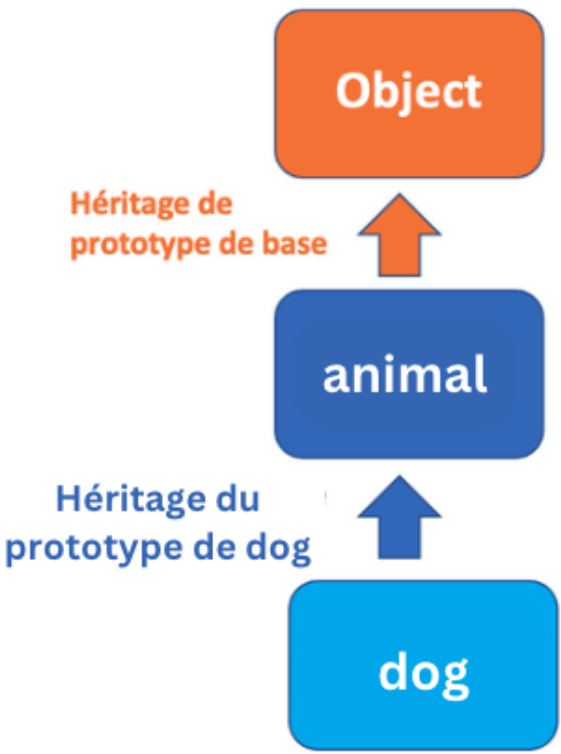
```
dog.bark();    // Wouf!
dog.breathe(); // Je respire.
```

- **dog.bark()** : Lorsque cette méthode est appelée, JavaScript la trouve directement dans l'objet dog, car elle y est définie. Il affiche donc le message "Wouf!" dans la console.

- **dog.breathe()** : Lorsque nous appelons cette méthode,

JavaScript commence par chercher la méthode breathe() dans l'objet dog. Ne trouvant pas cette méthode sur dog, JavaScript remonte dans la chaîne des prototypes et trouve la méthode breathe() dans l'objet animal, qui est le prototype de dog.

La méthode breathe() est ensuite exécutée, affichant "Je respire." dans la console.



En résumé, JavaScript permet cet héritage flexible grâce aux prototypes, permettant à un objet d'utiliser des méthodes d'un autre objet (son prototype) lorsqu'il ne trouve pas la méthode directement sur lui-même.
Cet exemple montre la puissance du système de prototypes en JavaScript, qui permet un partage flexible des méthodes entre objets sans avoir besoin de définir explicitement chaque méthode sur chaque objet.

Fonctions constructrices et prototypes : Création

d'objets à l'aide de fonctions constructrices

Avant l'introduction des classes en ES6, JavaScript utilisait des fonctions constructrices pour créer des objets similaires à des "classes". Les fonctions constructrices sont en réalité des fonctions ordinaires, mais lorsqu'elles sont utilisées avec le mot-clé new, elles créent de nouvelles instances d'objets.

Chaque fonction constructrice possède une propriété spéciale appelée prototype, qui définit le prototype des objets créés par cette fonction.

Voici un exemple :

```
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }

    // Ajoutons une méthode au prototype de 'Person'
    Person.prototype.introduce = function () {
            console.log(`Bonjour, je m'appelle ${this.name} et
j'ai ${this.age} ans.`);
    };

    // Création d'une nouvelle instance
    const student = new Person('Bob', 22);
     student.introduce(); // Bonjour, je m'appelle Bob et j'ai
22 ans.
```

Explication :

Fonction constructrice Person :

- Le premier bloc de ce code concerne la création de la fonction Person qui est une fonction **constructrice**. En JavaScript, il est courant de nommer les fonctions constructrices avec une **majuscule** (comme les classes) pour indiquer qu'elles sont destinées à être utilisées avec le mot-clé **new** pour créer des objets.
- À l'intérieur de la fonction, le mot-clé **this** fait référence à l'objet en cours de création. Dans cet exemple, chaque

fois que tu crées une nouvelle instance de **Person**, elle aura deux propriétés : **name** et **age**, qui seront définies à partir des arguments passés à la fonction.

Ajout d'une méthode au prototype de Person :

- Le deuxième bloc de code ajoute une méthode **introduce()** au **prototype** de la fonction Person (`Person.prototype.introduce`).
- Tous les objets créés avec cette fonction constructrice (comme student) partageront cette méthode.
- Le prototype est un mécanisme central en JavaScript. Chaque fonction (et donc chaque fonction constructrice) a une propriété **prototype**, qui est un **objet** utilisé pour définir des **méthodes** et des **propriétés** que toutes les instances créées à partir de cette fonction peuvent partager.
- Dans la méthode **introduce()**, this fait référence à l'instance actuelle de Person. Ainsi, **this.name** et **this.age** accèdent aux propriétés name et age de l'objet spécifique.

Création d'une instance de Person et appel de la méthode introduce()

- Le dernier bloc de ce code crée une instance (objet) à partir de la fonction Person. Cet objet (appelé student) aura une propriété name égale à 'Bob' et une propriété age égale à 22.
- Lorsque tu appelles **new Person(...)**, JavaScript crée un nouvel **objet vide**, et la fonction constructrice est exécutée, initialisant les propriétés name et age.

La dernière ligne appelle la méthode **introduce()** sur l'instance student. Cette méthode a été ajoutée au **prototype** de **Person**, donc elle est partagée par toutes les instances créées à partir de **Person**.
Lorsque ce code est exécuté, la console de votre navigateur affichera le résultat suivant :

```
Bonjour, je m'appelle Bob et j'ai 22 ans.
```

Récapitulatif :

- **Fonction constructrice** : Utilisée pour créer des objets. Ici, Person crée des objets avec les propriétés name et

age.

- **Prototype** : Un moyen d'ajouter des méthodes partagées à tous les objets créés à partir d'une fonction constructrice. Ici, la méthode introduce() est partagée par tous les objets de type Person.
- **Objet student** : Un nouvel objet créé à partir de Person, avec ses propres propriétés name et age, mais partageant la méthode introduce() du prototype.

Différence entre les classes et les prototypes : Pourquoi JavaScript reste orienté prototype sous le capot ?

Avec l'introduction d'ES6, JavaScript a introduit la syntaxe des classes, qui donne une apparence plus familière aux développeurs venant de langages comme Java ou Python. Cependant, il est important de noter que les classes en JavaScript sont essentiellement une abstraction du modèle basé sur les prototypes. Sous le capot, JavaScript utilise toujours des prototypes pour gérer l'héritage et la création d'objets.

Comme nous l'avions étudié dans les chapitres précédents, voici un exemple de classe en ES6 :

```
    class Person {
        constructor(name, age) {
            this.name = name;
            this.age = age;
        }

        introduce() {
             console.log(`Bonjour, je m'appelle ${this.name} et
j'ai ${this.age} ans.`);
        }
    }

    const teacher = new Person('Alice', 40);
      teacher.introduce(); // Bonjour, je m'appelle Alice et
j'ai 40 ans.
```

Même si la syntaxe de classe est utilisée ici, JavaScript crée en réalité

un lien de prototype entre **teacher** et **Person.prototype**. La classe n'est qu'un enrobage syntaxique du même mécanisme basé sur les prototypes.

Différences principales :

- Les prototypes sont la façon fondamentale dont JavaScript gère l'héritage et la délégation de méthodes. Chaque **fonction constructrice** a un **prototype**, et chaque **objet** a un lien vers son **prototype**.
- Les **classes** fournissent une syntaxe **plus lisible** et plus **structurée**, mais sous le capot, elles utilisent le même mécanisme que les prototypes.
- En résumé, bien que la syntaxe des classes rende JavaScript plus proche des langages orientés classe, le langage reste basé sur les prototypes et continue d'utiliser ce modèle en **interne**.

Conclusion

Les prototypes sont un concept fondamental en JavaScript qui sous-tend la manière dont les objets partagent des propriétés et des méthodes. Bien que JavaScript ait introduit une syntaxe de classe plus moderne avec ES6, le langage reste intrinsèquement basé sur un modèle orienté prototype. Chaque objet en JavaScript a un lien vers un prototype, et c'est à travers ce lien que se réalise l'héritage dans le langage.

Dans ce chapitre, nous avons exploré :

- Ce qu'est un prototype et comment il sert de mécanisme d'héritage et de délégation dans JavaScript.
- L'héritage par prototypes, où les objets peuvent hériter les uns des autres en suivant une chaîne de prototypes.
- Les fonctions constructrices et leur lien avec les prototypes, offrant un aperçu de la création d'objets avant l'introduction des classes modernes.
- Enfin, nous avons vu la différence entre les classes et les prototypes, en expliquant que bien que la syntaxe des classes simplifie la création d'objets pour les développeurs, JavaScript repose toujours sur le mécanisme de prototypes sous le capot.

Maîtriser les prototypes est essentiel pour comprendre comment JavaScript gère l'héritage et la réutilisation du code. Que vous utilisiez

des fonctions constructrices, des objets littéraux, ou des classes, le prototype est toujours présent et joue un rôle central dans la manière dont les objets se comportent et interagissent. En comprenant et en maîtrisant cette mécanique, vous pourrez écrire du code plus optimisé, réutilisable et maintenable.

Chapitre 13

Le DOM (Document Object Model)

Imaginez que votre page web est comme un immense puzzle interactif. Chaque morceau de ce puzzle – qu'il s'agisse d'un paragraphe, d'une image, ou d'un bouton – est une pièce essentielle. Pour manipuler ce puzzle, nous avons besoin d'un "plan" qui nous permet d'accéder à chaque pièce, de la déplacer, la modifier ou même la faire disparaître. C'est ici qu'entre en scène le **DOM** (Document Object Model).

Le DOM est une interface qui nous offre une représentation structurée de notre page web. Il transforme le document HTML en un arbre d'éléments que nous pouvons manipuler à l'aide de JavaScript. Grâce au DOM, nous pouvons modifier la structure, le style et le contenu d'une page web à la volée, sans avoir à recharger tout le site. Cela donne vie à nos pages, les rendant dynamiques, réactives et interactives.

Dans ce chapitre, nous plongerons dans le monde fascinant du DOM pour apprendre comment interagir avec lui, en maîtrisant les bases qui transformeront vos sites statiques en véritables applications interactives.

Introduction au DOM et son utilisation en JavaScript

Maintenant que vous maîtrisez les bases du langage JavaScript, il est temps de découvrir comment l'utiliser pour concevoir des pages web dynamiques et interactives. Pour cela, l'exploration du **DOM (Document Object Model)** est incontournable. Ce dernier permet de manipuler les éléments, attributs et textes d'une page web en temps réel. Dans ce chapitre, nous allons explorer ensemble les ressources et méthodes pratiques que JavaScript et le DOM mettent à notre disposition pour créer des pages modernes et réactives.

Qu'est-ce que le DOM ?

Le **DOM (Document Object Model)** est une représentation en arbre des éléments d'une page HTML ou XML. En d'autres termes, il s'agit d'une modélisation structurée de la page web, où chaque élément (balises, attributs, textes) devient un nœud dans cet arbre. Grâce à JavaScript, il est possible d'interagir avec ces nœuds pour modifier le contenu, ajouter ou supprimer des éléments, et gérer des événements dynamiques.

Le DOM n'est pas spécifique à JavaScript, il sert d'interface de programmation standardisée, prise en charge par tous les navigateurs modernes (Chrome, Firefox, Safari, Edge, etc.), permettant d'accéder aux éléments de la page et de les manipuler indépendamment du langage de programmation utilisé.

Pourquoi le DOM est-il crucial en JavaScript ?

Le DOM est fondamental en **JavaScript** car il constitue le lien entre le code et la page web affichée. Grâce à cette interface, JavaScript peut modifier l'apparence, le contenu, et le comportement des pages sans avoir à les recharger. Voici quelques-unes des raisons pour lesquelles le DOM est si important pour les développeurs :

- **Accès aux éléments dynamiques** : Le DOM permet de sélectionner et manipuler n'importe quel élément HTML de la page, facilitant ainsi la modification du contenu en fonction des interactions de l'utilisateur.
- **Gestion des événements** : Grâce au DOM, JavaScript peut réagir aux actions des utilisateurs (clics, soumissions de formulaires, déplacements de souris, etc.) et déclencher des fonctions spécifiques.
- **Animations et style dynamique** : Le DOM permet également de modifier les styles CSS en direct, ce qui ouvre la voie à la création d'animations fluides et d'effets visuels captivants.
- **Interaction avec les données** : En utilisant des techniques comme Ajax, il est possible de charger des données dynamiquement depuis un serveur sans recharger la page entière.

En résumé, le DOM est l'élément clé qui rend possible la création de pages web interactives et réactives, en permettant à JavaScript d'accéder facilement aux éléments de la page et de les manipuler en fonction des besoins.

Comment fonctionne le DOM ?

Le DOM fonctionne en représentant la structure HTML d'une page sous forme d'un arbre d'objets. Chaque élément, attribut ou texte devient un **nœud** de cet arbre, que JavaScript peut manipuler via des méthodes spécifiques. Lorsqu'un navigateur charge une page web, il analyse le HTML et génère cet arbre DOM. Chaque nœud de l'arbre est alors un objet JavaScript doté de propriétés et de méthodes permettant de le manipuler.

Par exemple, lorsqu'un utilisateur clique sur un bouton, le DOM capte cet événement et déclenche la fonction JavaScript associée, qui peut alors réagir en modifiant le contenu, l'apparence ou même la structure de la page.

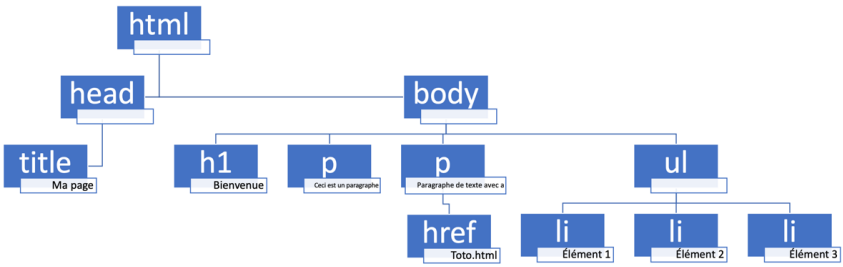
Le DOM est donc la passerelle entre la structure statique d'une page HTML et la puissance dynamique de JavaScript.

Exemple simple :

Une manière simple de comprendre le DOM est de l'imaginer comme un arbre structuré représentant les documents HTML. Cet arbre est illustré dans l'exemple de code HTML et la figure ci-dessous.

```
<!DOCTYPE html>
<html>

<head>
    <title>Ma page</title>
</head>

<body>
    <h1>Bienvenue</h1>
    <p>Ceci est un paragraphe.</p>
    <p>Paragraphe de texte avec a:
        <a href="toto.html">lien</a> ici.
    </p>
    <ul>
        <li>Élément 1</li>
        <li>Élément 2</li>
        <li>Élément 3</li>
    </ul>
```

```
</body>

</html>
```



Explication de l'arbre DOM :

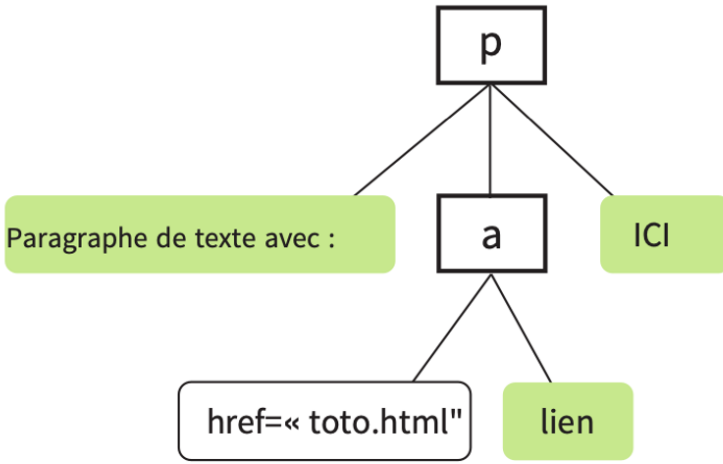
- **html** : C'est la racine de l'arbre DOM, représentant l'élément <html>.
- **head** et **body** : Les éléments <head> et <body> sont les enfants directs de l'élément <html>.
- **title** : Cet élément est un enfant de <head> et contient le texte "Ma page".
- <**body**> a plusieurs enfants : <h1>, deux <p>, et un <ul>.
- Le premier <**p**> contient le texte "Ceci est un paragraphe".
- Le deuxième <**p**> contient un lien <**a**> avec l'attribut href="toto.html" et le texte "lien".
- L'élément <**ul**> contient deux éléments <**li**>, "Élément 1", "Élément 2" et "Élément 2". Dans cet arbre, chaque nœud représente un élément HTML, un attribut, ou un contenu textuel. JavaScript peut accéder à ces nœuds et les manipuler via le DOM.

L'élément, ses attributs et son contenu sont tous des nœuds dans l'arbre de nœuds du DOM.

Par exemple :

```
<p>Paragraphe de texte avec a:
    <a href="toto.html">lien</a> ici.
 </p>
```

Cet élément **p** ci-dessous peut être représenté par l'arbre partiel suivant :



Grâce au DOM, JavaScript dispose de toute la flexibilité nécessaire pour générer du contenu HTML dynamique. Voici ce qu'il permet de faire :

Modifier l'ensemble des éléments HTML présents sur la page.
Changer ou mettre à jour les attributs HTML de n'importe quel élément.
Manipuler les styles CSS de façon dynamique pour personnaliser l'apparence de la page.
Supprimer ou ajouter des éléments et des attributs HTML existants.
Insérer de nouveaux éléments ou attributs HTML pour enrichir la structure de la page.
Réagir aux événements HTML existants (comme les clics, les survols, etc.) pour déclencher des actions.
Créer et gérer de nouveaux événements HTML afin d'interagir plus efficacement avec les utilisateurs.

Conclusion

Dans ce chapitre, nous avons exploré les bases du DOM (Document Object Model) et découvert comment JavaScript peut l'utiliser pour manipuler les éléments, les attributs, et le contenu d'une page web de manière dynamique. Le DOM joue un rôle central dans la création de pages interactives, en permettant de réagir aux actions des utilisateurs et de transformer une page statique en une expérience fluide et réactive.

Mais ce n'est que le début. Dans les chapitres qui suivent, nous allons

approfondir cette connaissance en explorant :

- Les méthodes spécifiques pour sélectionner et modifier des éléments du DOM de manière plus efficace.
- Comment gérer et manipuler les événements pour rendre vos pages web véritablement interactives.
- Des exemples concrets et des cas d'utilisation réels pour maîtriser pleinement la manipulation du DOM.

Avec ces nouvelles compétences, vous serez en mesure de créer des applications web plus sophistiquées et dynamiques. Préparez-vous à plonger encore plus profondément dans le cœur de JavaScript et du DOM !

Chapitre 14

Sélection d'éléments dans le DOM

Après avoir découvert les bases du DOM dans les chapitres précédents, il est temps de passer à une étape essentielle dans la création d'interactions dynamiques avec une page web : la sélection d'éléments. Le DOM est une structure complexe qui peut contenir des dizaines, voire des centaines d'éléments, chacun ayant un rôle spécifique dans l'affichage et le comportement de la page. Mais pour pouvoir manipuler ces éléments avec JavaScript, il est crucial de savoir comment les sélectionner efficacement.

Dans les chapitres à venir, nous allons explorer en détail les différentes façons de manipuler les éléments HTML au sein du DOM. Toutefois, avant de pouvoir modifier, ajouter ou supprimer un élément, il est essentiel de savoir comment cibler et accéder à cet élément. Pour ce faire, plusieurs méthodes de sélection sont disponibles :

- Cibler un élément par son identifiant unique.
- Rechercher des éléments partageant la même classe.
- Sélectionner des éléments en fonction de leur balise HTML.
- Utiliser des sélecteurs CSS avancés pour un ciblage précis.
- Manipuler des collections d'objets HTML.

Chacune de ces méthodes offre des avantages particuliers, en fonction des besoins spécifiques de votre projet. Nous allons les découvrir en profondeur pour que vous puissiez choisir la technique la plus adaptée dans chaque situation.

À titre d'exemple, nous commencerons par examiner une simple page HTML et apprendrons à sélectionner ses éléments avec efficacité. Cela vous permettra de prendre un contrôle total sur le DOM et d'ouvrir la voie à des manipulations avancées qui rendront vos pages web interactives et réactives.

Afin d'explorer les différentes méthodes de sélection des éléments dans le DOM, nous allons nous appuyer sur le code HTML suivant, représentatif d'une page web index ;html par exemple:

```
    <!DOCTYPE html>
    <html>

    <head>
        <meta charset="utf-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1">
        <title>Formation JavaScript</title>
    </head>

    <body>
        <h1 id="title1">Formation JavaScript</h1>
        <div class="introduction">
            <h2 id="intro">Introduction au DOM</h2>
             <p>Voici le paragraphe 1 avec du texte introductif
1</p>
             <p>Voici le paragraphe 2 avec du texte introductif
2</p>
             <p>Voici le paragraphe 3 avec du texte introductif
3</p>
        </div>
        <div class="introduction">
            <p id="paraUrl">Paragraphe de description</p>
            <a href="https://www.google.com/">lien google</a>
        </div>

        <div class="description">
                    <p>Utiliser le DOM pour rendre vos pages
interactives</p>
        </div>

        <div id="summary">
            <p>Voici le paragraphe 1 avec du résumé</p>
            <p>Voici le paragraphe 2 avec du résumé</p>
        </div>
```

```
        <script src="my_script.js"></script>
    </body>
    </html>
```

Cette page contient plusieurs éléments HTML tels que des titres, des paragraphes, et des liens. Nous allons explorer comment les sélectionner à l'aide de JavaScript en fonction de différents critères (identifiant, classe, balise, etc.).

Sélection d'éléments HTML par identifiant

Lorsque l'élément que vous cherchez possède un identifiant, la méthode la plus simple pour le sélectionner dans le DOM est d'utiliser **getElementById()**. L'identifiant (ou id) est unique sur une page web, il n'y a donc qu'un seul élément associé à cet identifiant.

Dans l'exemple suivant, nous sélectionnons l'élément <**h2**> qui possède l'id "**intro**".

```
    const myTitle = document.getElementById("intro");
     console.log(myTitle); // Affiche l'élément <h2> avec l'id
"intro"
```

Ce code à mettre dans le fichier my_script.js.

Explication : Nous avons utilisé getElementById() pour récupérer l'élément ayant l'identifiant "intro". Le résultat est stocké dans la variable myTitle, et la commande console.log(myTitle) permet de vérifier que l'élément a bien été récupéré.

En lançant la page index.html et en ouvrant la console de votre navigateur via les raccourcis clavier, vous verrez s'afficher l'élément HTML ayant l'identifiant "intro".

```
      <h2 id="intro">Introduction au DOM</h2>
```

De la même manière, nous pouvons récupérer l'élément avec l'id "paraUrl" :

```
    const myParaUrl = document.getElementById("paraUrl");
     console.log(myParaUrl); // Affiche l'élément <p> avec l'id
"paraUrl"
```

Explication : Ici, nous sélectionnons le paragraphe avec l'id "paraUrl" et l'affichons dans la console pour confirmation.

```
<p id="paraUrl">Paragraphe de description</p>
```

Sélection d'éléments HTML par nom de classe

Il est possible de sélectionner tous les éléments qui partagent une même **classe** avec la méthode **getElementsByClassName()**. Dans notre exemple, plusieurs éléments partagent la classe "introduction". Cette méthode retourne une collection d'éléments.

```
const introElements = document.getElementsByClassName("introduction");
console.log(introElements); // Affiche une collection des éléments avec la classe "introduction"
```

Explication : Nous avons utilisé **getElementsByClassName()** pour récupérer tous les éléments qui partagent la classe "introduction". Le résultat est une collection (ou tableau d'éléments), stockée dans la variable introElements.

Résultat :

Vous remarquez que la collection de la classe « introduction » contient deux éléments. Pour accéder à chaque élément individuellement, nous utilisons un index :

```
console.log(introElements[0]); // Affiche le premier élément avec la classe "introduction"
console.log(introElements[1]); // Affiche le deuxième élément avec la classe "introduction"
```

Explication : Les éléments dans une collection sont accessibles via un **index** (comme pour un tableau). Ici, **introElements[0]** sélectionne le premier élément de la collection, et **introElements[1]** le deuxième.

Lorsque vous exécutez ce code à travers le lancement de votre page index.html, vous obtiendrez l'affichage dans la console de ces deux éléments qui partagent la classe « introduction » :

```
▼<div class="introduction">
    <h2 id="intro">Introduction au DOM</h2>
    <p>Voici le paragraphe 1 avec du texte introductif 1</p>
    <p>Voici le paragraphe 2 avec du texte introductif 2</p>
    <p>Voici le paragraphe 3 avec du texte introductif 3</p>
  </div>
▼<div class="introduction">
    <p id="paraUrl">Paragraphe de description</p>
    <a href="https://www.google.com/">lien google</a>
  </div>
```

Sélection d'éléments HTML par nom de balise

Pour récupérer tous les éléments d'un même type de balise, nous utilisons **getElementsByTagName()**. Cela permet de sélectionner, par exemple, tous les éléments de la page.

```
    const divElements = document.getElementsByTagName("div");
      console.log(divElements);  // Affiche une collection de
tous les éléments <div>
```

Explication : Nous avons utilisé **getElementsByTagName()** pour sélectionner toutes les divisions ( ) de la page. L'utilisation de **getElementsByTagName()** nécessite de spécifier le nom de la balise à sélectionner. Dans cet exemple, nous avons choisi de sélectionner les éléments ayant la balise « div »

Le résultat est une collection d'éléments.

```
my_script.js:2
HTMLCollection(4) [div.introduction, div.introduction, div.description,
div#summary, summary: div#summary] i
  0: div.introduction
  1: div.introduction
  2: div.description
  3: div#summary
  summary: div#summary
  length: 4
```

Le résultat montre qu'il y'a quatre élément avec la balise . Pour accéder à chaque division individuellement :

```
const divElements = document.getElementsByTagName("div");
console.log(divElements);  // Affiche une collection de tous
les éléments <div>

console.log(divElements[0]); // Affiche la première division
console.log(divElements[1]); // Affiche la deuxième division
console.log(divElements[2]); // Affiche la troisième division
console.log(divElements[3]); // Affiche la troisième division
```

Explication : Comme pour les classes, nous pouvons accéder à chaque élément par son index dans la collection.

Lorsque vous lancez votre page index.html, vous obtiendrez, dans la console de votre navigateur, le résultat suivant :

```
▶ <div class="introduction">…</div>
▶ <div class="introduction">…</div>
▶ <div class="description">…</div>
▶ <div id="summary">…</div>
```

De la même manière, pour sélectionner tous les paragraphes (<p>) :

```
const paragraphElements =
document.getElementsByTagName("p");
    console.log(paragraphElements); // Affiche une collection
de tous les éléments <p>
```

Explication : Ici, nous sélectionnons tous les éléments <p> de la page. Le résultat est une collection d'éléments que nous pouvons

inspecter via la console.

Résultat :

```
▶ HTMLCollection(7) [p, p, p, p#paraUrl, p, p, p, paraUrl: p#paraUrl]
```

Supposons maintenant que nous souhaitions sélectionner uniquement les paragraphes dans la division avec l'id "summary" :

```
const summaryDiv = document.getElementById("summary");
const summaryParagraphs = summaryDiv.getElementsByTagName("p");
console.log(summaryParagraphs); // Affiche une collection des paragraphes enfants du <div> avec l'id "summary"
```

Explication : Nous récupérons d'abord la division avec l'id "**summary**" en utilisant getElementById(), puis, en appelant **getElementsByTagName()** sur cet élément, nous récupérons uniquement les paragraphes enfants de cette division. Ainsi, le résultat affiché dans la console montre qu'il existe seulement deux paragraphes dans la division avec l'id « summary » :

```
▼ HTMLCollection(2) [p, p] i
  ▶ 0: p
  ▶ 1: p
    length: 2
```

Sélection des éléments par les sélecteurs CSS

Nous pouvons également utiliser les sélecteurs CSS pour sélectionner des éléments dans le DOM avec **querySelector()** et **querySelectorAll()**.

Exemple 1 : Sélection par classe

```
const introElement = document.querySelector(".introduction");
console.log(introElement); // Affiche le premier élément avec la classe "introduction"
```

Explication : Avec **querySelector()**, nous avons sélectionné le **premier élément** qui possède la classe "introduction". Contrairement à **getElementsByClassName()**, cette méthode ne retourne qu'un seul élément.

Si nous souhaitons sélectionner tous les éléments avec la classe "introduction", nous utilisons **querySelectorAll()** :

```
const allIntroElements = document.querySelectorAll(".introduction");
console.log(allIntroElements); // Affiche une NodeList de tous les éléments avec la classe "introduction"
```

Explication : **querySelectorAll()** retourne tous les éléments correspondant au sélecteur CSS spécifié en argument, ici tous les éléments avec la classe "**introduction**". Remarquez le point (.) qui précède le nom de la classe (.introduction).

Pour sélectionner tous les paragraphes dans les divisions avec la classe "introduction" :

```
const introParagraphs = document.querySelectorAll(".introduction p");
console.log(introParagraphs); // Affiche tous les paragraphes dans les div avec la classe "introduction"
```

Explication : Dans ce code, nous avons sélectionné tous les éléments <**p**> qui se trouvent dans une division avec la classe "introduction".

Exemple 2 : Sélection par identifiant

```
const summaryElement = document.querySelector("#summary");
console.log(summaryElement); // Affiche l'élément avec l'id "summary"
```

Explication : Nous avons utilisé **querySelector()** pour sélectionner l'élément ayant l'identifiant "summary". Remarquez que le symbole # (dièse) qui précede le nom de l'identifiant
Cette méthode peut être utilisée pour n'importe quel sélecteur CSS.

Sélection d'éléments HTML par des collections d'objets HTML

Certains éléments HTML comme le <body>, les formulaires, les images, et les liens peuvent être accédés directement via des collections prédéfinies.

Sélection du body :

```
const myBody = document.body;
console.log(myBody); // Affiche l'élément <body>
```

Explication : Nous avons récupéré l'élément <body> directement avec la propriété **document.body**.

Sélection du titre de la page :

```
const pageTitle = document.title;
 console.log(pageTitle); // Affiche le titre de la page
"Formation JavaScript"
```

Explication : Ici, nous avons utilisé la propriété **document.title** pour accéder au titre de la page.

Sélection des formulaires :

Si notre page contient des formulaires, nous pouvons les récupérer avec **document.forms** :

```
const forms = document.forms;
 console.log(forms); // Affiche une collection de tous les
formulaires
```

Explication : **document.forms** retourne une collection de tous les formulaires présents sur la page. Nous pouvons accéder à chacun d'eux en utilisant des index.

Pour cibler un formulaire spécifique :

```
console.log(forms[1]); // Affiche le deuxième formulaire
```

Explication : Nous avons récupéré le deuxième formulaire de la page en accédant à son index dans la collection.

Si nous voulons accéder à un champ spécifique, comme le champ avec name="email" :

```
    console.log(forms[1].email); // Affiche l'élément <input> avec le name="email"
    console.log(forms[1].email.value); // Affiche la valeur du champ "email"
```

Explication : Nous avons accédé au champ d'input "email" du deuxième formulaire et récupéré sa valeur avec la propriété .value.

Sélection des liens :

```
    const links = document.links;
    console.log(links); // Affiche une collection de tous les liens (<a>)
```

Explication : **document.links** retourne tous les éléments <a> (liens) de la page sous forme de collection.

Pour récupérer un lien spécifique et afficher sa valeur href :

```
    console.log(links[0].href); // Affiche l'URL du premier lien
```

Explication : Nous avons sélectionné le premier lien de la page et affiché la valeur de son attribut href.

Conclusion

Dans ce chapitre, nous avons exploré les différentes méthodes permettant de sélectionner des éléments dans le DOM. Que ce soit via l'identifiant, la classe, le nom de balise, ou encore à l'aide de sélecteurs CSS, JavaScript nous offre une flexibilité incroyable pour cibler précisément n'importe quel élément d'une page web.

Ces techniques sont essentielles pour manipuler le contenu d'une page

de manière dynamique. Grâce à elles, vous pouvez désormais non seulement accéder aux éléments du DOM, mais aussi vous préparer à les modifier, les styliser, ou réagir à des événements utilisateur.

Dans les chapitres suivants, nous allons approfondir cette base en explorant comment interagir plus en détail avec les éléments sélectionnés, afin de rendre nos pages encore plus interactives et réactives.

Chapitre 15

Modification, ajout et suppression d'éléments du DOM

Après avoir appris à sélectionner des éléments dans le DOM, l’étape suivante consiste à comprendre comment interagir avec ces éléments de manière plus dynamique. En effet, la véritable puissance de JavaScript réside dans sa capacité à modifier le contenu d'une page web en temps réel, sans rechargement.

Dans ce chapitre, nous allons explorer comment modifier les éléments du DOM, que ce soit pour changer leur contenu textuel, leurs attributs, ou leurs styles CSS. Nous verrons également comment supprimer des éléments, soit en les effaçant entièrement du document, soit en supprimant uniquement certains de leurs enfants. Ces manipulations sont cruciales pour rendre une page web interactive et réactive aux actions de l'utilisateur.

Pour cela, nous continuerons à nous baser sur l'exemple HTML utilisé dans le chapitre précédent, et nous utiliserons les méthodes JavaScript adaptées pour rendre nos pages plus vivantes et dynamiques

Modification d'éléments du DOM

Dans le chapitre précédent, nous avons appris à sélectionner des éléments du DOM en utilisant différentes méthodes. Maintenant que nous savons comment cibler ces éléments, nous allons voir comment les manipuler, que ce soit pour modifier leur contenu, leurs attributs ou même pour les supprimer entièrement du document.

Pour ce chapitre, nous allons continuer à nous appuyer sur l'exemple HTML (index ;html) utilisé précédemment, afin d'illustrer comment modifier et supprimer les éléments du DOM de manière dynamique avec JavaScript.

Exemple de structure HTML :

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
          <meta   name="viewport"   content="width=device-width,
initial-scale=1">
    <title>Formation JavaScript</title>
</head>

<body>
    <h1 id="title1">Formation JavaScript</h1>
    <div class="introduction">
        <h2 id="intro">Introduction au DOM</h2>
         <p>Voici le paragraphe 1 avec du texte introductif 1</
p>
         <p>Voici le paragraphe 2 avec du texte introductif 2</
p>
         <p>Voici le paragraphe 3 avec du texte introductif 3</
p>
    </div>

    <div class="introduction">
        <p id="paraUrl">Paragraphe de description</p>
        <a href="https://www.google.com/">lien google</a>
```

```
    </div>

    <div class="description">
                <p>Utiliser le DOM pour rendre vos pages
interactives</p>
    </div>

    <div id="summary">
        <p>Voici le paragraphe 1 avec du résumé</p>
        <p>Voici le paragraphe 2 avec du résumé</p>
    </div>

    <script src="my_script.js"></script>
</body>

</html>
```

Une fois que nous avons sélectionné un élément, nous pouvons facilement modifier son contenu, ses attributs, ou encore son style. Pour cela, JavaScript met à disposition plusieurs propriétés et méthodes.

Modification du texte d'un élément

La méthode la plus simple pour changer le contenu textuel d'un élément est d'utiliser la propriété textContent.

Exemple : Modifier le titre h2

Ajoutons le code suivant dans le fichier my_script.js :

```
    const myTitle = document.getElementById("intro");
     myTitle.textContent = "Présentation du DOM"; // Change le
texte du titre
     console.log(myTitle.textContent); // Affiche "Présentation
du DOM"
```

Explication : Dans cet exemple, nous avons sélectionné l'élément <h2> avec l'id "intro" et le stocker dans la variable myTitle. Ensuite, nous avons modifié son contenu texte grâce à **textContent**. Cette

propriété remplace tout le texte à l'intérieur de l'élément.

Lorsque vous lancez votre page index.html et vous ouvrez la console de votre navigateur, vous obtiendrez résultat suivant qui affiche bien le nouveau contenu de l'élément <h2> ayant pour id « intro ».

```
Présentation du DOM
```

Modification du contenu HTML d'un élément

Si nous voulons insérer du HTML à l'intérieur d'un élément (comme des balises , , etc.), nous utilisons la propriété innerHTML.

Exemple : Ajouter un lien dans un paragraphe

Dans cet exemple, nous allons modifier le contenu du paragraphe ayant l'id "paraUrl" pour y ajouter un lien hypertexte. Nous utilisons la propriété innerHTML, qui permet de manipuler directement le contenu HTML de l'élément sélectionné.

```
    const myParaUrl = document.getElementById("paraUrl");
     myParaUrl.innerHTML = 'Paragraphe de description avec un
<a href="https://www.example.com">nouveau lien</a>';
    console.log(myParaUrl.innerHTML);
```

Explication :

- Nous avons d'abord récupéré l'élément <p> avec l'id "paraUrl" en utilisant getElementById().
- Ensuite, nous avons modifié son contenu en lui injectant du HTML grâce à la propriété innerHTML. Nous avons ajouté un lien hypertexte <a> qui pointe vers "https://www.example.com" et qui affiche "nouveau lien".

Pour vérifier que la modification a bien été effectuée, nous avons ajouté la dernière ligne de code pour voir le contenu modifié du paragraphe dans la console. Cette commande affichera dans la console le nouveau contenu HTML du paragraphe, y compris le lien ajouté :

```
Paragraphe de description avec un <a href="https://www.example.com">nouveau lien</a>
```

Résumé : En utilisant innerHTML, vous pouvez modifier le contenu HTML interne d'un élément. Ici, nous avons ajouté un lien au paragraphe existant et avons vérifié le résultat en inspectant l'élément dans la console du navigateur et en visualisant les modifications dans la page web.

Modification des attributs d'un élément

Pour modifier les attributs d'un élément HTML, nous utilisons la méthode **setAttribute().** Cela permet de changer des attributs comme **href**, **src**, **class**, ou tout autre attribut valide d'un élément HTML.

Exemple : Modifier le lien Google

Dans cet exemple, nous modifions l'attribut **href** du premier lien de la page pour qu'il pointe vers un autre site, et nous changeons également le texte du lien.

```
const myLink = document.querySelector("a");
myLink.setAttribute("href", "https://www.example.com");
myLink.textContent = "lien vers example";
```

Explication :

- setAttribute() permet de modifier l'attribut href pour rediriger le lien vers "https://www.example.com".
- Ensuite, nous avons changé le texte affiché dans le lien avec textContent.

Pour vérifier la modification de l'attribut **href**, vous pouvez ajouter la commande suivante utilisant la méthode **getAttribute**() :

```
console.log(myLink.getAttribute("href"));
```

Résultat :

```
https://www.example.com
```

Vous pouvez également vérifier le texte du lien :

```
console.log(myLink.textContent);
```

Résultat :

```
https://www.example.com
lien vers example
```

Vous pouvez vérifier dans le navigateur : Sur la page, le lien apparaîtra désormais comme suit :

lien vers example

Modification des styles CSS d'un élément dans le DOM

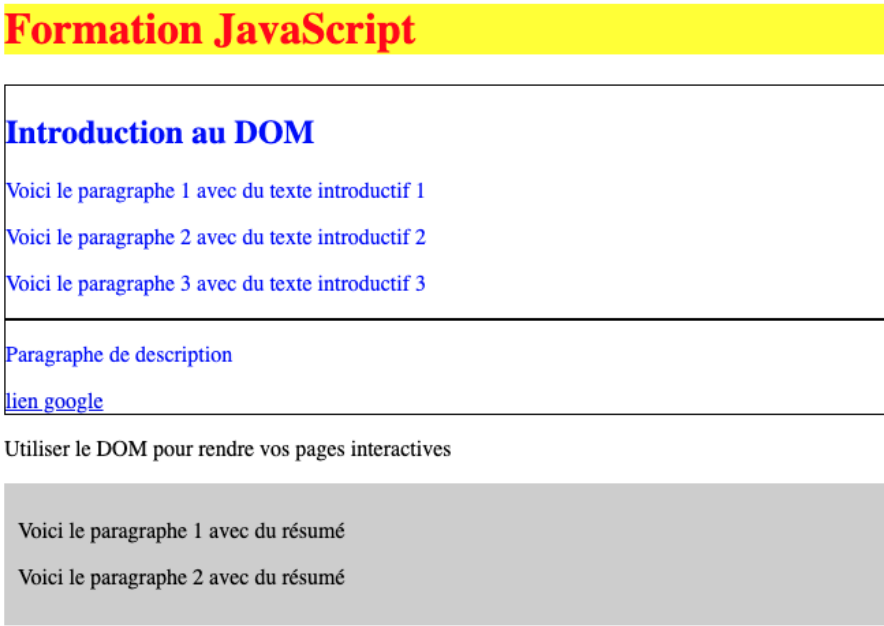
Dans cette section, nous allons aborder comment modifier les styles CSS d'un élément directement à partir de JavaScript. Manipuler les styles dynamiquement permet de rendre les pages web plus interactives, en fonction des actions des utilisateurs ou des événements spécifiques. Pour cela, nous allons nous baser sur un fichier CSS que nous avons créé, nommé my_style.css, et notre fichier HTML déjà utilisé dans ce chapitre. Nous lions ce fichier HTML à ce fichier css en ajoutant entre les balises <head> et </head> le code suivant :

```
<link rel="stylesheet" type="text/css" href="my_style.css">
```

Exemple de fichier CSS : my_style.css

```
/* Style par défaut pour le titre */
#title1 {
    color: red;
    background-color: yellow;
}

/* Style pour la classe introduction */
.introduction {
    font-size: 16px;
    color: blue;
    border: 1px solid black;
}

/* Style pour la division summary */
#summary {
    background-color: lightgray;
    padding: 10px;
}
```

À ce stade, le fichier JavaScript **my_script.js** est vide, si vous ouvrez, dans votre navigateur, la page index.html, vous obtiendrez le résultat :



Vous remarquez que les styles spécifiés dans le fichier CSS my_style.css ont été pris en compte par la page HTML index.html.

Modifier la couleur d'un élément

Pour modifier la couleur d'un élément HTML avec JavaScript, nous utilisons la propriété **style.color**. Cela permet de changer la couleur du texte.

Exemple : Changer la couleur du titre <h1>

Ajoutons le code suivant dans le script my_script.js :

```
    const myTitle = document.getElementById("title1");
      myTitle.style.color = "green"; // Change la couleur du
titre en vert
```

Explication :

- Nous avons d'abord sélectionné l'élément <**h1**> avec **l'id "title1"**.

- Ensuite, nous avons utilisé la propriété **style.color** pour changer la couleur du texte en vert. Cette modification écrase la règle CSS existante dans my_style.css qui définissait la couleur en rouge.

Lorsque vous actualisez votre page index.html, vous obtiendrez :

Vous pouvez constater que la couleur du titre <h1> est passée du rouge au vert.

Modifier la couleur de l'arrière-plan d'un élément

La propriété **style.backgroundColor** permet de changer la couleur d'arrière-plan d'un élément.

Exemple : Changer la couleur de fond de la division summary

Ajoutons le code suivant au fichier my_script.js

```
    const summaryDiv = document.getElementById("summary");
     summaryDiv.style.backgroundColor = "lightblue"; // Change
la couleur de fond en bleu clair
```

Explication :

- Nous avons sélectionné la division avec l'id "**summary**".
- Nous avons modifié la couleur d'arrière-plan en bleu clair avec la propriété **style.backgroundColor**. Cela remplace la couleur définie dans le fichier CSS (qui était "lightgray").

L'actualisation de votre page index.html vous permet d'obtenir le résultat :

Vous pouvez constater que la couleur de la division d'id « summary » est passée du gris au bleu clair.

Modifier le style de plusieurs éléments à la fois

Il est possible de modifier le style de plusieurs éléments en même temps, surtout lorsqu'ils partagent une classe commune. Utilisons **getElementsByClassName()** pour sélectionner tous les éléments avec la classe "introduction" et modifier leur couleur de texte.

Exemple : Changer la couleur de tous les éléments avec la classe "introduction"

Ajoutons le code suivant au fichier my_script.js :

```
const introElements = document.getElementsByClassName("introduction");
for (let i = 0; i < introElements.length; i++) {
    introElements[i].style.color = "purple"; // Change la couleur de texte en violet pour tous les éléments
}
```

Ce code modifie la couleur de texte de tous les éléments HTML qui ont la classe "introduction". Voici une explication détaillée:

Ligne 1 : const introElements = document.getElementsByClassName("introduction");

- Cette ligne utilise la méthode getElementsByClassName() pour obtenir tous les éléments du DOM (Document Object Model) qui ont la classe CSS "introduction".
- document.getElementsByClassName("introduction") retourne une collection HTML (une sorte de tableau) contenant tous les éléments qui possèdent cette classe.
- Cette collection est stockée dans la variable introElements.

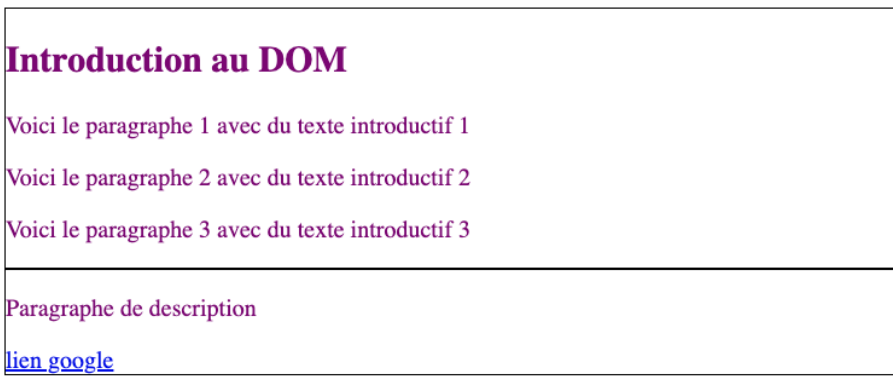
La boucle for:

- C'est une boucle for qui itère tous les éléments de la collection introElements.
- À chaque itération de la boucle, cette ligne sélectionne

un élément dans la collection introElements en utilisant introElements[i].

- introElements[i].style.color permet de modifier la propriété CSS color de cet élément. On lui assigne la valeur "purple", ce qui change la couleur du texte de cet élément en violet.

La classe « introduction » du fichier HTML (index.html) est composée de 6 éléments (Un titre h2, 4 paragraphes p et un lien a). Lorsque vous actualisez votre page index.html, vous obtiendrez le résultat :



Avec une boucle for, nous avons appliqué un nouveau style à chacun de ces éléments, modifiant la couleur du texte en violet.

Ajouter une bordure à un élément

Pour ajouter ou modifier une bordure autour d'un élément, nous utilisons la propriété **style.border**.

Exemple : Ajouter une bordure au titre <h1>

Ajoutons le code suivant dans le fichier my_script.js :

```
    myTitle.style.border = "2px solid black"; // Ajoute une
bordure noire autour du titre
```

Explication :

Nous avons ajouté une bordure **noire solide** de **2 pixels** autour du titre <**h1**>. Si l'élément possédait déjà une bordure dans le fichier CSS, celle-ci sera remplacée.

Si vous actualisez votre page index, vous obtiendrez le résultat :

Utiliser les méthodes setProperty() et removeProperty()

La méthode **setProperty()** permet de définir ou de remplacer une propriété CSS sur un élément de manière plus détaillée. Elle prend trois arguments : la propriété CSS, la valeur, et une priorité (comme "important").
La méthode **removeProperty()** supprime une propriété CSS spécifique d'un élément.

Exemple : Ajouter une propriété CSS avec **setProperty()**

Nous allons modifier la taille de la police du titre ayant l'id « title ». Nous avons déjà sélectionné ce titre dans la première ligne du code du fichier my_script.js. Ajoutons le code suivant :

```
                myTitle.style.setProperty("font-size",    "60px",
"important"); // Change la taille du texte avec priorité
```

Explication :

setProperty() modifie la taille de la police du titre en "60px", et applique la priorité "**important**", ce qui permet de surcharger toute autre règle CSS.

Lorsque vous actualisez la page index.html, vous obtiendrez le résultat :

Vous pouvez observer que la taille du titre a considérablement augmenté par rapport à sa taille initiale.

Supprimer une propriété CSS avec removeProperty()

Exemple : supprimer la bordure du titre h1 :

Ajoutons le code suivant dans my_script.js :

```
myTitle.style.removeProperty("border"); // Supprime la bordure autour du titre
```

Explication :

removeProperty() supprime la propriété "border" du titre, ce qui fait disparaître la bordure noire ajoutée précédemment.

Résultat :

Formation JavaScript

En résumé, dans cette section, nous avons vu comment manipuler les styles CSS à l'aide de JavaScript. En modifiant les propriétés de style directement dans le DOM, nous pouvons rendre nos pages web plus interactives et adaptatives. Que ce soit pour changer la couleur d'un élément, son arrière-plan, ou encore pour ajouter une bordure, JavaScript permet de contrôler l'apparence des éléments de manière dynamique et en temps réel. Les méthodes **setProperty()** et **removeProperty()** offrent une flexibilité supplémentaire pour gérer les styles plus précisément.

Suppression des éléments du DOM

Suppression d'un élément avec remove()

Voyons maintenant comment supprimer un élément du DOM à l'aide de la méthode **remove()**.

```
    const myPara = document.getElementById("paraUrl");
    myPara.remove(); // Supprime l'élément du DOM
```

Explication : En utilisant remove(), nous avons supprimé l'élément

<p> avec l'id "paraUrl" du DOM.

Si vous essayez de récupérer l'élément après l'avoir supprimé, le résultat sera null, car l'élément n'existe plus dans le DOM. Voici le code mis à jour :

```
const myPara = document.getElementById("paraUrl");
myPara.remove(); // Supprime l'élément du DOM

console.log(myPara); // Affiche l'élément avant suppression
console.log(document.getElementById("paraUrl")); // Affichera
"null" après suppression
```

Explication :

- Dans la première commande, nous sélectionnons l'élément <p> ayant l'identifiant "paraUrl". La méthode getElementById() permet de récupérer cet élément dans le DOM, et nous stockons le résultat dans la variable myPara. À ce stade, l'élément est toujours présent dans le DOM.
- Dans la deuxième ligne, nous utilisons la méthode **remove()** sur l'élément sélectionné pour le supprimer du DOM. Après l'appel à remove(), l'élément <p> avec l'id "paraUrl" n'existe plus dans la structure du DOM et n'apparaît donc plus sur la page.
- La troisième ligne de code affiche l'élément myPara dans la console. L'élément existe toujours en tant qu'objet JavaScript dans la mémoire, même après sa suppression du DOM, car il a été stocké dans la variable myPara avant sa suppression.
- Enfin, après la suppression de l'élément du DOM, si vous essayez de récupérer à nouveau l'élément avec getElementById("paraUrl"), cette commande retournera null, car l'élément n'existe plus dans la structure DOM. Cela indique que l'élément a bien été supprimé du document.

En résumé, myPara contient toujours l'élément supprimé, mais uniquement en tant qu'objet JavaScript, il n'est plus dans le DOM.
document.getElementById("paraUrl") renverra **null**, car l'élément a

été retiré de la structure de la page.

Suppression d'un enfant avec removeChild()

Enfin, nous allons voir comment supprimer un enfant d'un élément parent en utilisant la méthode removeChild().

```
    const summaryDiv = document.getElementById("summary");
                            const       firstParagraph       =
summaryDiv.getElementsByTagName("p")[0];
       summaryDiv.removeChild(firstParagraph);  // Supprime  le
premier paragraphe de la division summary
```

Explication :

Nous avons sélectionné la division avec l'id "summary".
Ensuite, nous avons récupéré le premier paragraphe à l'intérieur de cette division et l'avons supprimé en utilisant removeChild().

Après avoir supprimé le paragraphe, vous pouvez vérifier le contenu restant dans la division "summary" en entrant la commande suivante :

```
    console.log(summaryDiv.innerHTML);
```

Vous devriez voir seulement le deuxième paragraphe restant dans la division avec l'id « summary »

Sur la page, le premier paragraphe à l'intérieur de la division "summary" aura disparu, laissant uniquement le second paragraphe.

Ajouter des éléments au DOM

Dans cette section, nous allons voir comment ajouter des éléments au DOM en utilisant JavaScript. Il est souvent nécessaire d'ajouter dynamiquement du contenu à une page web, comme un nouveau paragraphe, une nouvelle division, ou tout autre type d'élément. Pour cela, nous utilisons plusieurs méthodes qui permettent de créer de nouveaux éléments et de les insérer à des endroits spécifiques dans la structure du DOM.

Les principales étapes pour ajouter un élément sont :

- Créer un nouvel élément HTML.

- Ajouter du contenu à cet élément (texte, HTML, attributs).
- Insérer cet élément dans le DOM à un endroit souhaité.

Nous allons utiliser les méthodes suivantes :

- document.createElement() : pour créer un nouvel élément.
- appendChild() et insertBefore() : pour insérer un élément dans le DOM

Création d'un élément avec createElement()

La méthode createElement() nous permet de créer un nouvel élément HTML, comme une division, un paragraphe, ou une balise de lien.

Exemple : Création d'un paragraphe

```
const newParagraph = document.createElement("p");
```

Explication :

Ici, nous créons un nouvel élément <p> (paragraphe) et nous le stockons dans la variable newParagraph. À ce stade, l'élément est créé, mais il n'est pas encore visible dans le DOM, car il n'a pas encore été inséré.

Ajout de contenu à l'élément

Une fois l'élément créé, nous pouvons y ajouter du contenu. Cela peut être du texte, un autre élément HTML, ou des attributs (comme une classe ou un identifiant).

Exemple : Ajouter du texte à l'élément

```
  newParagraph.textContent = "Ceci est un nouveau paragraphe ajouté dynamiquement.";
```

Explication :

Nous utilisons la propriété textContent pour insérer du texte dans le nouvel élément <p>. Maintenant, le paragraphe a du contenu, mais il n'est toujours pas visible dans la page, car il n'a pas encore été ajouté au DOM.

Insertion de l'élément dans le DOM avec appendChild()

Pour rendre l'élément visible sur la page, il doit être inséré à un endroit précis du DOM. La méthode appendChild() permet d'ajouter l'élément en tant que dernier enfant d'un élément parent.

Exemple : Ajouter le nouveau paragraphe à la fin de la division "summary"

```
const summaryDiv = document.getElementById("summary");
summaryDiv.appendChild(newParagraph);
```

Explication :

Nous avons sélectionné la division avec l'id "summary", qui sert d'élément parent.
Ensuite, nous avons ajouté le paragraphe newParagraph comme dernier enfant de cette division avec appendChild().

Pour vérifier que l'élément a bien été ajouté au DOM, vous pouvez inspecter le contenu de la division "summary" après l'insertion. Il suffit d'ajouter la ligne de code suivante :

```
console.log(summaryDiv.innerHTML);
```

Voici le code complet :

```
    const newParagraph = document.createElement("p");
    newParagraph.textContent = "Ceci est un nouveau paragraphe
ajouté dynamiquement.";
    const summaryDiv = document.getElementById("summary");
    summaryDiv.appendChild(newParagraph);
    console.log(summaryDiv.innerHTML);
```

Dans la console de votre navigateur, vous obtiendrez l'affichage des trois paragraphes de la division « summary ». Voici le résultat :

```
        <p>Voici le paragraphe 1 avec du résumé</p>
        <p>Voici le paragraphe 2 avec du résumé</p>
    <p>Ceci est un nouveau paragraphe ajouté dynamiquement.</p>
```

Sur la page, vous devriez voir apparaître le nouveau paragraphe avec le texte "Ceci est un nouveau paragraphe ajouté dynamiquement." dans la division "summary".

Insertion avant un autre élément avec insertBefore()

Il est également possible d'insérer un élément à une position spécifique avant un autre élément avec la méthode **insertBefore()**.

Exemple : Ajouter le nouveau paragraphe avant le premier paragraphe de la division "summary"

```
  const newParagraph = document.createElement("p");
   newParagraph.textContent = "Ceci est un nouveau paragraphe
ajouté dynamiquement.";
  const summaryDiv = document.getElementById("summary");

   const firstParagraph = summaryDiv.getElementsByTagName("p")
[0];
  summaryDiv.insertBefore(newParagraph, firstParagraph);
```

Explication :

- Ligne 1 : Nous avons créé un nouvel élément paragraphe et l'avons stocké dans la variable newParagraph.
- Ligne 2 : Nous avons ajouté du texte à ce nouveau paragraphe.
- Ligne 3 : Nous avons sélectionné l'élément avec l'id "summary".
- Ligne 4 : Nous avons récupéré le premier paragraphe dans la division "summary" pour savoir où insérer le nouveau paragraphe.
- Ligne 5 : Nous avons inséré le nouveau paragraphe avant le premier paragraphe de la division en utilisant la méthode insertBefore().

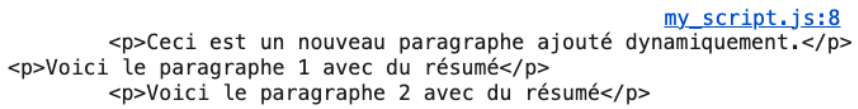
Pour vérifier que l'élément a bien été ajouté au DOM, vous pouvez inspecter le contenu de la division "summary" après l'insertion. Il suffit d'ajouter à la fin du code précèdent la ligne de code suivante :

```
    console.log(summaryDiv.innerHTML);
```

- Avant l'exécution du script, la division "summary" contient deux paragraphes.
- Après l'exécution du script, un nouveau paragraphe sera ajouté juste avant le premier paragraphe, avec le texte :

"Ceci est un nouveau paragraphe ajouté dynamiquement."

Résultat :



Remarque :

Nous avons appris comment créer et ajouter des éléments HTML au DOM, mais vous pourriez vous demander pourquoi ne pas simplement le faire directement en HTML, ce qui semble plus rapide. La réponse est que l'objectif de JavaScript n'est pas de remplacer le travail d'HTML, mais plutôt de rendre les éléments HTML plus dynamiques et interactifs.

Grâce à JavaScript, il est possible d'exécuter certaines actions en fonction de conditions spécifiques, de répéter des tâches en boucle, ou encore de modifier des éléments en réponse aux interactions de l'utilisateur. Par exemple, on peut afficher ou cacher des éléments en fonction de l'état d'une page, générer des listes automatiquement en parcourant des données, ou encore déclencher des événements en fonction des actions des utilisateurs. C'est cette flexibilité et cette capacité à réagir en temps réel qui rendent JavaScript si puissant lorsqu'il est utilisé pour manipuler le DOM.

Ajout des éléments au DOM : Nouvelles méthodes

Nous avons vu comment utiliser createElement() pour créer un nouvel élément, puis comment l'insérer dans le DOM à l'aide des méthodes appendChild() et insertBefore(). Cependant, il existe d'autres méthodes modernes qui facilitent encore davantage l'ajout d'éléments dans le DOM : after(), before(), append() et prepend(). Ces méthodes sont intuitives et permettent d'insérer des éléments à l'intérieur ou autour d'éléments existants.

Utilisation des méthodes after() et before()

Ces deux méthodes permettent d'ajouter des éléments après ou avant un élément existant, mais en dehors de celui-ci.

Exemple : Ajouter un paragraphe avant et après un titre h1

```
    const myTitle = document.getElementById("title1");

    // Ajout après le titre
       myTitle.after(document.createElement('p').textContent  =
"Paragraphe après le titre");

    // Ajout avant le titre
      myTitle.before(document.createElement('p').textContent  =
"Paragraphe avant le titre");
```

Explication :

Ligne 1 du code: cette ligne sélectionne l'élément <h1> de la page ayant l'identifiant "title1". Cet élément est ensuite stocké dans la variable myTitle, ce qui nous permet de le manipuler plus tard dans le script.

Ligne 2 du code:

- **document.createElement('p')** crée un nouvel élément <p> (paragraphe).
- **.textContent = "Paragraphe après le titre"** ajoute du texte à ce nouvel élément paragraphe. Le texte "Paragraphe après le titre" sera donc le contenu du nouveau paragraphe.
- **myTitle.after(...)** ajoute ce nouveau paragraphe juste après l'élément sélectionné myTitle, donc après le titre <h1>.

Résultat : Un paragraphe sera ajouté après le titre <h1> avec le texte "Paragraphe après le titre".

Ligne 3 du code :

- De la même manière que précédemment, **document.createElement('p')** crée un nouveau paragraphe.
- **.textContent = "Paragraphe avant le titre"** insère du texte dans ce nouveau paragraphe.
- **myTitle.before(...)** insère ce paragraphe juste avant l'élément myTitle, donc avant le titre <h1>.

Résultat : Un paragraphe sera ajouté avant le titre <h1> avec le

texte "Paragraphe avant le titre".

Utilisation des méthodes append() et prepend()

Ces méthodes permettent d'ajouter des éléments à l'intérieur d'un élément existant.

- append() insère un élément à la fin des enfants de l'élément.
- prepend() insère un élément au début des enfants.

Exemple : Ajouter du contenu à la division summary

```
    const myDiv = document.getElementById("summary");

    // Ajout à la fin
        myDiv.append(document.createElement('p').textContent  =
"Paragraphe ajouté à la fin");

    // Ajout au début
       myDiv.prepend(document.createElement('p').textContent  =
"Paragraphe ajouté au début");
```

Explication :

- append() ajoute un nouvel élément à la fin des enfants de la division "summary".
- prepend() ajoute un nouvel élément au début des enfants de la même division.

Pourquoi utiliser ces méthodes ?
Ces méthodes sont plus simples et plus intuitives que appendChild() et insertBefore(). Elles offrent des solutions rapides pour ajouter des éléments sans avoir besoin de manipuler manuellement des nœuds dans le DOM.

Conclusion

Dans ce chapitre, nous avons exploré les méthodes fondamentales pour interagir avec le DOM en JavaScript, un aspect crucial pour dynamiser les pages web et créer des interfaces utilisateur réactives.

Grâce aux différentes techniques pour modifier, ajouter, et supprimer des éléments du DOM, vous êtes désormais en mesure de manipuler le contenu d'une page de manière dynamique, en réponse aux actions des utilisateurs ou aux changements de données.

Que ce soit à travers des méthodes comme appendChild(), removeChild(), ou les plus récentes telles que before() et after(), JavaScript offre une flexibilité sans égale pour construire des interfaces interactives. Nous avons également vu comment ajuster le style des éléments et modifier leurs attributs pour améliorer leur apparence et leur comportement.

La maîtrise de ces outils vous permettra de créer des applications web plus riches et plus interactives, en rendant vos pages plus dynamiques et engageantes. Ces compétences sont essentielles pour tout développeur souhaitant créer des interfaces modernes, performantes et réactives.

Chapitre 16

Événements dans le DOM

Les événements sont au cœur de l'interactivité sur le web. Que ce soit un clic sur un bouton, une saisie de texte dans un champ, ou le survol d'un élément, chaque action de l'utilisateur déclenche un événement que le navigateur peut capturer et traiter. Comprendre comment manipuler ces événements est essentiel pour rendre une page web véritablement interactive.

Dans ce chapitre, nous allons explorer les différents types d'événements disponibles dans le DOM, et voir comment nous pouvons écouter ces événements et y réagir en utilisant JavaScript. Que vous souhaitiez afficher un message après un clic, valider un formulaire en temps réel, ou encore animer un élément lors d'un survol, la gestion des événements dans le DOM vous permettra de créer des expériences utilisateur dynamiques et engageantes.

Plongeons dans le monde fascinant des événements dans le DOM pour apprendre à capturer et à gérer les interactions de l'utilisateur de manière fluide et efficace !

Introduction

Les événements en JavaScript représentent des actions ou des occurrences qui se produisent dans une page web. Qu'il s'agisse du chargement de la page, d'un clic sur un bouton, de la soumission d'un formulaire, ou de la saisie de texte dans un champ, tous ces événements permettent aux développeurs d'interagir de manière dynamique avec les éléments HTML de la page. Cela leur permet de créer des fonctionnalités réactives et interactives pour les utilisateurs.

Il existe plusieurs catégories d'événements en JavaScript, comme les événements liés à la souris, au clavier, aux formulaires, ou encore au chargement d'une page. Chaque événement a un nom spécifique et est associé à une fonction qui sera exécutée lorsque cet événement se déclenche. En réalité, la plupart des interactions que nous effectuons sur une page web sont des événements. Toutefois, il n'est pas nécessaire de connaître chaque événement par cœur pour être un bon développeur, car il en existe une multitude. Ce qui est important, c'est de comprendre le principe et de savoir comment utiliser ces événements.

Parmi les événements les plus courants, on peut citer :

- L'événement click : pour détecter un clic sur un élément.
- L'événement dblclick : pour détecter un double-clic.
- L'événement mousemove : pour suivre le mouvement de la souris.
- L'événement mouseover : lorsque la souris survole un élément.
- L'événement mouseout : lorsque la souris quitte un élément.
- L'événement keypress : pour détecter l'appui d'une touche.
- L'événement keyup : pour détecter le relâchement d'une touche.
- L'événement resize : lors du redimensionnement de la fenêtre.
- L'événement scroll : pour suivre le défilement de la page.
- En ce qui concerne les formulaires, on retrouve également des événements essentiels :

- L'événement input : pour capter les modifications dans un champ de saisie.
- L'événement submit : lors de la soumission d'un

formulaire.

- L'événement reset : pour détecter la réinitialisation d'un formulaire.
- L'événement change : lorsque la valeur d'un élément de formulaire change.
- L'événement focus : lorsque le focus est donné à un champ.
- L'événement blur : lorsque le focus quitte un champ.

Pour illustrer ces événements populaires, nous allons nous appuyer sur le code HTML suivant (index.html) et le fichier de style CSS (my_style.css).

Index.html :

```
    <!DOCTYPE html>
    <html>

    <head>
        <meta charset="utf-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1">
        <title>Formation JavaScript</title>
                    <link rel="stylesheet" type="text/css"
href="my_style.css">
    </head>

    <body>
        <img id="mylogo" src="images/logo.png">
        <h1 id="title1">Formation JavaScript</h1>
        <div class="introduction">
            <h2 id="intro">Introduction au DOM</h2>
             <p>Voici le paragrpahe 1 avec du texte introductif
1</p>
             <p>Voici le paragrpahe 2 avec du texte introductif
2</p>
             <p>Voici le paragrpahe 3 avec du texte introductif
3</p>
        </div>
        <button class="btn">Cliquer ici</button>
```

```
    <script src="my_script.js"></script>
</body>

</html>
```

my_style.css:

```
/* Style par défaut pour le titre */
#title1 {
    color: red;
    background-color: yellow;
}

/* Style pour la classe introduction */
.introduction {
    font-size: 16px;
    color: blue;
    border: 1px solid black;
}

/* Style pour la division summary */
#summary {
    background-color: lightgray;
    padding: 10px;
}

.btn {
    background-color: #F95E05;
    font-size: 16px;
    padding: 12px 34px;
    font-weight: 600;
    height: 3.5em;
    color: #FFF;
    text-transform: uppercase;
    text-align: center;
    text-decoration: none;
    border: 1px solid transparent;
    border-radius: 7px;
```

```
        cursor: pointer;
    }
```

À ce stade, si vous lancez votre page index.html, vous obtiendrez un résultat qui ressemble au résultat ci-dessous (ne pas oublier de mettre une petite image nommée logo.png dans le répertoire du projet).

Formation JavaScript

Introduction au DOM

Voici le paragrpahe 1 avec du texte introductif 1

Voici le paragrpahe 2 avec du texte introductif 2

Voici le paragrpahe 3 avec du texte introductif 3

CLIQUER ICI

Événement click

Commençons par l'événement click, l'un des événements les plus couramment utilisés. Dans cet exemple, nous allons modifier la couleur de l'arrière-plan d'une division lorsque l'utilisateur clique sur celle-ci.

Étape 1 : Sélectionner l'élément à modifier
Tout d'abord, nous sélectionnons la division ayant la classe **.introduction**. Utilisons pour cela la méthode querySelector(). Ajoutons le code suivant dans le script my_script.js :

```
let myDivision = document.querySelector(".introduction");
```

Étape 2 : Modifier la couleur de l'élément

Pour illustrer simplement, nous pouvons immédiatement changer la couleur d'arrière-plan de la division myDivision en rouge. Ajoutons le code suivant dans my_script.js :

```
myDivision.style.backgroundColor = "red";
```

Si vous actualisez votre page index, vous consteriez que cette modification est immédiate et ne dépend pas d'une **action de l'utilisateur**. Ce que nous voulons, c'est que cette modification se produise **uniquement** lorsque l'utilisateur **clique** sur la division ayant la classe **.introduction**. Pour cela, nous allons utiliser un **gestionnaire d'événements**.

Ajouter un gestionnaire d'événements click

Pour déclencher une action au moment où l'utilisateur clique sur la division, nous devons ajouter un gestionnaire d'événements. Le gestionnaire **addEventListener()** est la méthode la plus utilisée et **recommandée** pour écouter les événements en JavaScript. Il fonctionne en trois étapes :

- Écoute de l'événement.
- Détection de l'événement lorsqu'il se produit.
- Exécution de l'action associée à cet événement.

Exemple de base : Changer la couleur de fond lors du clic
Nous allons maintenant ajouter un gestionnaire d'événements **click** à la division **myDivision**. Lorsqu'un utilisateur clique dessus, la couleur de fond passera au vert.

Remplaçons alors le code ajouté à l'étape 2 par le code suivant :

```
myDivision.addEventListener("click", function () {
    myDivision.style.backgroundColor = "green";
});
```

Explication :

addEventListener("click", function() {...}) : Ce code écoute

l'événement **click** sur la division. Lorsqu'un clic est détecté sur la zone de la division myDivision, la fonction anonyme change la couleur d'arrière-plan en vert.

Différentes façons d'ajouter des événements

Il existe plusieurs manières de passer une fonction dans **addEventListener()**. Voyons ces options :

- **Méthode 1 : Utilisation d'une fonction anonyme**

La première méthode consiste à définir directement la fonction à l'intérieur de **addEventListener().** C'est la méthode la plus courante et c'est la méthode que nous avions utilisé pour changer la couleur d'arrière plan de la division myDivision.

```
myDivision.addEventListener("click", function () {
    myDivision.style.backgroundColor = "green";
});
```

Explication :

- **myDivision.addEventListener()** est une méthode qui permet d'attacher un écouteur d'événement à un élément du DOM. Ici, myDivision est une division sélectionnée dans le DOM, probablement via document.querySelector() ou document.getElementById().
- Le premier paramètre "**click**" indique que l'événement à écouter est un clic. Ainsi, chaque fois que l'utilisateur clique sur l'élément myDivision, l'événement est déclenché.
- Le deuxième paramètre est une **fonction anonyme** qui sera exécutée lorsque l'événement de clic sera détecté.
- La ligne de code à l'intérieur de la fonction anonyme change la propriété backgroundColor (couleur de fond) de l'élément myDivision.
- Ainsi, lorsque l'utilisateur clique sur myDivision, la couleur d'arrière-plan de la division devient verte ("**green**").

- **Méthode 2 : Utilisation d'une fonction fléchée**

On peut également utiliser une **fonction fléchée**, une syntaxe plus **concise** introduite en ES6.

```
    myDivision.addEventListener("click", () => {
        myDivision.style.backgroundColor = "green";
    });
```

Explication :

- Le premier paramètre "click" indique que nous voulons écouter un événement de clic. Cela signifie que chaque fois que l'utilisateur cliquera sur cette division, le gestionnaire d'événement (la fonction) sera exécuté.
- Le deuxième paramètre est une **fonction fléchée** (arrow function) qui spécifie ce qui doit se passer lorsque l'événement est détecté.
- À l'intérieur de la fonction fléchée, nous avons la ligne myDivision.style.backgroundColor = "green";. Cette ligne modifie le style CSS de la division en question. Plus précisément, elle change la couleur d'arrière-plan de la division en vert (green).
- Dès que l'utilisateur clique sur l'élément myDivision, cette action est exécutée, modifiant la couleur de fond.

- **Méthode 3 : Utilisation d'une fonction nommée**

Enfin, vous pouvez définir une **fonction séparée**, puis l'utiliser comme gestionnaire d'événements.

```
function changeColor() {
    myDivision.style.backgroundColor = "green";
}

myDivision.addEventListener("click", changeColor);
```

- Ici, nous définissons d'abord une fonction nommée appelée **changeColor().** Cette fonction, lorsqu'elle est appelée, modifie la propriété CSS backgroundColor (couleur d'arrière-plan) de l'élément myDivision, en lui assignant la couleur verte ("green").

- La fonction est définie indépendamment de l'événement, ce qui signifie que nous pouvons l'utiliser à plusieurs reprises, à différents endroits dans notre code si nécessaire.

- addEventListener("click", changeColor) est utilisé pour écouter l'événement click sur l'élément myDivision. Cela signifie que lorsque l'utilisateur clique sur myDivision, la fonction changeColor() est automatiquement appelée.

- Remarquez que changeColor est passée comme référence à la fonction (sans parenthèses). Cela permet à JavaScript d'exécuter la fonction uniquement lorsque l'événement click est détecté, et non immédiatement au moment où l'écouteur est défini.

Pourquoi utiliser une fonction nommée ?

- **Réutilisabilité** : L'avantage d'utiliser une fonction nommée comme changeColor() est que cette fonction peut être réutilisée dans d'autres contextes. Par exemple, nous pourrions aussi utiliser cette même fonction pour un autre événement, comme un survol de souris.
- **Lisibilité** : Le code devient plus lisible car la logique de ce qui doit se passer au clic est séparée. Cela permet de mieux organiser le code, surtout si la logique devient plus complexe.

L'événement dblclick

L'événement **dblclick** se produit lorsqu'un utilisateur double-clique sur un élément. Cela peut être utile pour créer des interactions plus spécifiques.

Exemple : Double-cliquer pour changer la couleur d'un titre

HTML du titre :

```
<h1 id="titre1">Formation JavaScript</h1>
```

Mettons le code suivant dans le fichier my_script.js :

```
const title = document.getElementById("title1");
title.addEventListener('dblclick', function() {
    title.style.color = "green"; // Change la couleur du titre
```

```
en vert
});
```

Explication :

- Ici, document.getElementById("titre1") sélectionne l'élément HTML dont l'id est titre1. Dans notre exemple, il s'agit probablement d'un titre <h1>.
- Cet élément est ensuite stocké dans la variable title. Cela permet de manipuler cet élément dans le reste du code.
- Nous utilisons addEventListener() pour écouter un événement spécifique sur l'élément title. Ici, l'événement que nous écoutons est **dblclick**, qui se déclenche lorsque l'utilisateur effectue un double-clic sur l'élément.
- Le deuxième argument de addEventListener() est une fonction anonyme. Cette fonction sera exécutée chaque fois que l'événement **dblclick** sera détecté sur le titre.
- Lorsque l'utilisateur double-clique sur le titre, cette ligne de code modifie le style de l'élément title en changeant la propriété color (couleur du texte) en vert ("green").
- Cela signifie que, après un double-clic, le texte de l'élément title deviendra vert à l'écran.

Résumé du fonctionnement :

- **Sélection de l'élément** : Nous sélectionnons l'élément avec l'id title1.
- **Ajout d'un gestionnaire d'événement** : Nous attachons un gestionnaire pour l'événement dblclick à cet élément.
- **Exécution de l'action** : Lorsqu'un double-clic est détecté sur cet élément, la couleur du texte change pour devenir verte.

L'événement mousemove

L'événement **mousemove** est déclenché lorsque l'utilisateur déplace sa souris sur un élément. Il est souvent utilisé pour afficher des informations en fonction de la position de la souris.

Exemple : Afficher les coordonnées de la souris sur une image

HTML de l'image :

```
<img id="mylogo" src="logo.png">
```

JavaScript :

```
    const logo = document.getElementById("mylogo");
    logo.addEventListener('mousemove', function(event) {
            console.log("Position X : " + event.clientX + ",
Position Y : " + event.clientY);
    });
```

Explication :

- **document.getElementById("mylogo")** sélectionne l'élément HTML avec l'id mylogo. Dans cet exemple, il s'agit probablement d'une image ou d'un élément que nous voulons surveiller pour les mouvements de la souris.
- L'élément sélectionné est ensuite stocké dans la variable logo. Cela nous permet de manipuler cet élément plus facilement dans le code.
- **addEventListener()** est utilisé ici pour écouter l'événement **mousemove**, qui est déclenché chaque fois que l'utilisateur déplace la souris au-dessus de l'élément logo.
- Une fonction anonyme est passée comme deuxième argument à **addEventListener().** Cette fonction sera exécutée chaque fois que l'événement **mousemove** est détecté.
- Dans la fonction, nous utilisons **event.clientX** et **event.clientY** pour obtenir les coordonnées de la souris :
    - **event.clientX** représente la **position horizontale (X)** de la souris par rapport à la fenêtre du navigateur.
    - **event.clientY** représente la **position verticale (Y)** de la souris par rapport à la fenêtre du navigateur.

Ces informations sont ensuite affichées dans la console du navigateur avec **console.log(),** sous la forme : "Position X : [valeur], Position Y : [valeur]". Chaque fois que la souris se déplace sur l'élément, ces

coordonnées sont mises à jour et affichées.

```
Position X : 138, Position Y : 49
Position X : 143, Position Y : 136
Position X : 41, Position Y : 155
```

Les positions clientX, clientY, pageX, pageY, offsetX, et offsetY en JavaScript

Ces potions sont souvent utilisées dans le cadre des événements de souris en JavaScript.

clientX et clientY
Ces deux propriétés renvoient les coordonnées de la souris par rapport à la fenêtre visible du navigateur.

- **clientX** : La position horizontale de la souris (en pixels) à l'intérieur de la fenêtre visible du navigateur (viewport). Elle commence à 0 sur le bord gauche de la fenêtre et s'étend jusqu'au bord droit visible.
- **clientY** : La position verticale de la souris (en pixels) à l'intérieur de la fenêtre visible du navigateur (viewport). Elle commence à 0 sur le bord supérieur de la fenêtre et va jusqu'au bas de la partie visible.

Exemple d'utilisation : Imaginons que l'utilisateur déplace la souris sur une fenêtre du navigateur qui a été partiellement réduite. clientX et clientY donneront les coordonnées de la souris par rapport à cette zone visible, et non par rapport à l'ensemble du document ou de la page.

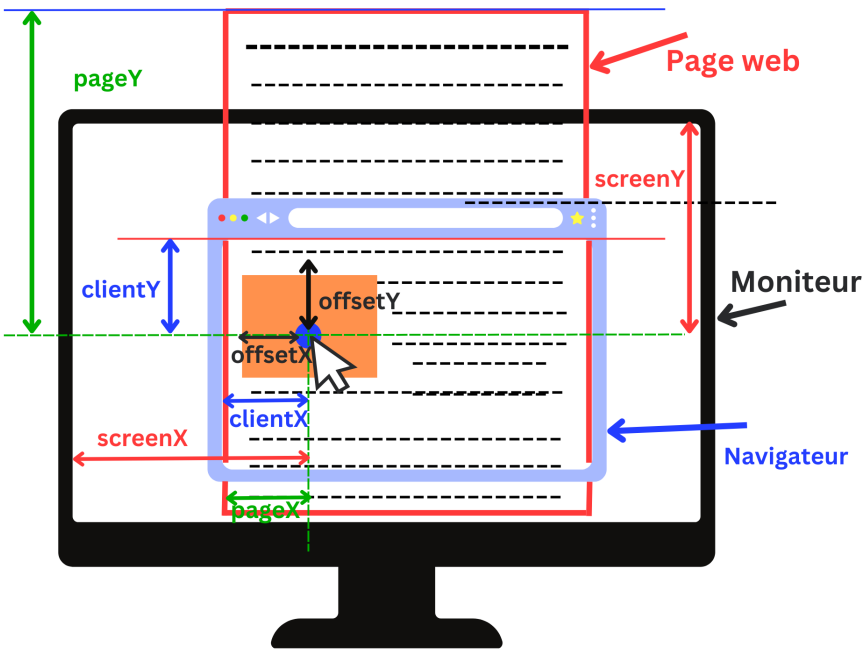
pageX et pageY

Ces propriétés renvoient les coordonnées de la souris par rapport à la page entière (le document), y compris les parties qui ne sont pas visibles en raison du défilement.

- **pageX** : La position horizontale de la souris (en pixels) par rapport au bord gauche de l'ensemble du document, même si une partie de la page a été défilée. Elle prend en compte tout le contenu, y compris celui qui se trouve en dehors de la zone visible.

- **pageY** : La position verticale de la souris (en pixels) par rapport au bord supérieur de l'ensemble du document. Elle inclut également la partie du document qui est en dehors de la zone visible, notamment si vous avez fait défiler vers le bas.

Exemple d'utilisation : Si l'utilisateur fait défiler la page vers le bas, les valeurs de pageX et pageY augmentent pour indiquer la position exacte de la souris dans le contexte de l'intégralité du document.



offsetX et offsetY
Ces propriétés renvoient les coordonnées de la souris par rapport à l'élément sur lequel l'événement de souris s'est produit.

- **offsetX** : La position horizontale de la souris à l'intérieur de l'élément déclenchant l'événement (en pixels). Elle commence à 0 sur le bord gauche de cet élément et s'étend jusqu'au bord droit de cet élément.

- **offsetY** : La position verticale de la souris à l'intérieur de l'élément déclenchant l'événement (en pixels). Elle commence à 0 sur le bord supérieur de cet élément et va jusqu'à son bord inférieur.

Exemple d'utilisation : Si vous avez une image ou une division (un conteneur), offsetX et offsetY vous indiqueront la position de la souris à l'intérieur de cet élément spécifique, peu importe où se trouve cet élément dans la page ou le document.

screenX et screenY
Ces propriétés renvoient les coordonnées de la souris par rapport à l'écran entier de l'ordinateur, et non pas juste par rapport à la fenêtre du navigateur ou à la page web.

- **screenX** : La position horizontale de la souris par rapport à l'écran. Cette valeur commence à 0 sur le bord gauche de l'écran et augmente à mesure que la souris se déplace vers la droite de l'écran.

- **screenY** : La position verticale de la souris par rapport à l'écran. Cette valeur commence à 0 sur le bord supérieur de l'écran et augmente à mesure que la souris se déplace vers le bas de l'écran.

Exemple d'utilisation : Si vous avez plusieurs fenêtres ou applications ouvertes, screenX et screenY vous donnent la position de la souris sur l'écran entier, pas uniquement sur une fenêtre spécifique ou une page web.

Comparaison entre ces propriétés :
clientX et **clientY** sont utilisés pour obtenir la position de la souris par rapport à la zone visible du navigateur (la fenêtre visible).

pageX et **pageY** donnent la position de la souris par rapport à l'ensemble du document, même les parties qui sont hors de la vue et doivent être atteintes par défilement.

offsetX et **offsetY** renvoient les coordonnées relatives à l'élément spécifique sur lequel l'événement a été déclenché.

L'événement mouseover et mouseout

Ces événements permettent de détecter quand la souris entre dans un élément (mouseover) et quand elle en sort (mouseout).

Exemple : Modifier la couleur de fond du bouton lors du survol et rétablir la couleur originale quand la souris sort

JavaScript :

```
    const button = document.querySelector('.btn');
    button.addEventListener('mouseover', function () {
          button.style.backgroundColor = "blue"; // Change la
couleur de fond en bleu
    });

    button.addEventListener('mouseout', function () {
         button.style.backgroundColor = "#F95E05"; // Rétablit
la couleur d'origine
    });
```

Explication :

- Ici, **document.querySelector('.btn')** est utilisé pour sélectionner l'élément HTML qui possède la classe .btn. Cet élément est ensuite stocké dans la variable button. L'élément sélectionné est un bouton (<button>) sur la page web.

- **addEventListener('mouseover', ...)** ajoute un gestionnaire d'événements pour écouter l'événement **mouseover**. Cet événement est déclenché lorsque l'utilisateur passe la souris sur l'élément (ici, le bouton).
- La fonction anonyme définie dans le second argument est exécutée dès que l'utilisateur survole le bouton avec la souris. Cette fonction modifie la propriété **backgroundColor** du style CSS de l'élément button, changeant la couleur de fond en bleu ("blue").

- **addEventListener('mouseout', ...)** ajoute un autre gestionnaire d'événements, cette fois pour l'événement **mouseout**. Cet événement est déclenché lorsque la souris **quitte** l'élément.
- Lorsque cet événement se produit, la fonction anonyme dans le gestionnaire d'événements rétablit la couleur de fond à sa valeur d'origine, #F95E05, une couleur orange. Cela réinitialise l'apparence du bouton lorsque la souris n'est plus au-dessus de l'élément.

En résumé :

- Lorsque l'utilisateur passe la souris sur le bouton (événement mouseover), la couleur de fond du bouton change en bleu.
- Lorsque l'utilisateur retire la souris du bouton (événement mouseout), la couleur de fond revient à l'orange initial #F95E05.

Ce type de comportement est couramment utilisé pour donner un effet visuel de survol sur les boutons, souvent vu dans les interfaces utilisateur pour améliorer l'interactivité et l'expérience utilisateur.

L'événement keypress, keydown et keyup

Les événements liés au clavier en JavaScript permettent de réagir à différentes actions de l'utilisateur lors de l'appui ou du relâchement des touches du clavier. Ils sont utiles dans de nombreux cas, comme la validation de formulaires en temps réel, la capture de raccourcis clavier ou la gestion d'actions spécifiques en fonction de la touche pressée.

Voici les trois événements principaux liés au clavier :

- **keydown** : Cet événement est déclenché lorsqu'une touche du clavier est enfoncée.
- **keypress** : Cet événement est déclenché lorsqu'une touche est enfoncée et que cette touche génère un caractère. Il est de moins en moins utilisé car keydown et keyup sont plus polyvalents.
- **keyup** : Cet événement est déclenché lorsque l'utilisateur relâche une touche du clavier.

L'événement keydown
L'événement keydown se produit lorsque n'importe quelle touche du clavier est enfoncée. Cet événement est déclenché dès que l'utilisateur commence à appuyer sur une touche, et il se produit avant que la touche ne soit relâchée.

Exemple :

```
document.addEventListener('keydown', function (event) {
    console.log('La touche pressée est : ' + event.key);
});
```

Explication :

- **document.addEventListener('keydown', ...)** attache un gestionnaire d'événement au document entier pour écouter les appuis de touches.
- **event.key** renvoie le caractère de la touche qui a été enfoncée. Par exemple, si l'utilisateur appuie sur la touche "A", la console affichera : "La touche pressée est : A".

Cela peut être utilisé pour surveiller n'importe quel appui de touche, que ce soit pour des raccourcis clavier, des jeux ou des actions spécifiques.

L'événement keypress
L'événement keypress se déclenche également lorsque l'utilisateur enfonce une touche, mais il se limite aux touches qui produisent des caractères imprimables (lettres, chiffres, symboles). Il ne fonctionne pas pour des touches comme Ctrl, Alt, Shift, ou les touches de direction.

Important : L'événement **keypress** est considéré comme obsolète dans certaines utilisations modernes. Il est recommandé d'utiliser **keydown** ou **keyup**.

Exemple :

```
document.addEventListener('keypress', function(event) {
    console.log('Caractère entré : ' + event.key);
});
```

Explication :

event.key dans ce cas renvoie uniquement les caractères imprimables. Si l'utilisateur appuie sur la touche "1", la console affichera "Caractère entré : 1".
Cet événement est encore utilisé pour certaines actions simples comme la saisie dans les champs de texte.

L'événement keyup
L'événement keyup se produit lorsque l'utilisateur relâche une touche après l'avoir enfoncée. C'est souvent utilisé pour réagir à la fin d'une saisie ou pour vérifier l'état final des touches après une interaction.

Exemple :

```
document.addEventListener('keyup', function(event) {
    console.log('La touche relâchée est : ' + event.key);
});
```

Explication :
keyup attend que l'utilisateur relâche la touche avant de déclencher l'action. Par exemple, si l'utilisateur appuie sur "E" et relâche cette touche, la console affichera "La touche relâchée est : E".

Cet événement est utile lorsque vous avez besoin de déclencher une action après que l'utilisateur a fini d'appuyer sur une touche, par exemple pour la validation en temps réel d'un formulaire.

Comparaison entre keydown, keypress et keyup

- **keydown** : Cet événement se déclenche dès que la touche est enfoncée. Il est compatible avec toutes les touches, y compris celles qui ne génèrent pas de caractères imprimables (comme Ctrl, Alt, Shift, etc.).

- **keypress** : Cet événement se déclenche lorsque l'utilisateur appuie sur une touche qui génère un caractère imprimable (lettres, chiffres, symboles). Il ne capte pas les touches de contrôle comme Ctrl, Alt, Shift. Cet événement est moins utilisé dans les projets modernes.

- **keyup** : Cet événement se déclenche uniquement lorsque l'utilisateur relâche une touche, et il fonctionne pour toutes les touches.

Exemples combinés
Exemple 1 : Changer la couleur du fond avec keydown et rétablir la couleur avec keyup
Cet exemple change la couleur de fond du document lorsqu'une touche est enfoncée et rétablit la couleur d'origine lorsque la touche est relâchée.

```
document.addEventListener('keydown', function () {
    document.body.style.backgroundColor = "lightblue";
});

document.addEventListener('keyup', function () {
    document.body.style.backgroundColor = "white";
```

```
    });
```

Explication :

- **keydown** modifie la couleur de fond en bleu clair dès que l'utilisateur commence à appuyer sur une touche.
- **keyup** rétablit la couleur de fond en blanc lorsque la touche est relâchée.

L'événement de redimensionnement de la fenêtre : resize
L'événement **resize** se produit lorsque la fenêtre du navigateur est redimensionnée.

Exemple : Afficher la taille de la fenêtre lors du redimensionnement

```
    window.addEventListener('resize', function () {
            console.log("Largeur : " + window.innerWidth + ",
Hauteur : " + window.innerHeight);
    });
```

- Ici, **window.addEventListener('resize', ...)** attache un gestionnaire d'événements à l'objet global window (la fenêtre du navigateur). L'événement **resize** est déclenché à chaque fois que la taille de la fenêtre du navigateur est modifiée par l'utilisateur (en la redimensionnant manuellement).
- L'écouteur d'événements capture l'action de redimensionnement et exécute la fonction associée chaque fois que l'événement **resize** se produit.

- **window.innerWidth** renvoie la largeur visible actuelle de la fenêtre du navigateur (en pixels). Elle inclut toute la partie de la fenêtre visible, à l'exclusion des barres de défilement.
- **window.innerHeight** renvoie la hauteur visible actuelle de la fenêtre du navigateur (en pixels), là encore sans compter les barres de défilement.
- À chaque fois que la fenêtre est redimensionnée, ces valeurs sont affichées dans la console sous la forme "Largeur : [valeur], Hauteur : [valeur]", permettant de suivre en temps réel les nouvelles dimensions de la

fenêtre.

Je vous invite à ouvrir votre page index.html, à afficher la console de votre navigateur, puis à redimensionner la fenêtre. Vous observerez alors que les dimensions actuelles de la fenêtre (largeur et hauteur) s'affichent dynamiquement dans la console.

L'événement de défilement : scroll

L'événement **scroll** est déclenché lorsque l'utilisateur fait défiler la page ou un élément qui peut être défilé.

Exemple : Afficher un message lorsque la page est défilée

```
window.addEventListener('scroll', function () {
    console.log("Page défilée");
});
```

Explication :

Cet événement capture chaque défilement de la page. Ici, nous affichons un simple message "Page défilée" dans la console à chaque fois que l'utilisateur fait défiler la page.

L'événement scroll se déclenche uniquement si la page a un contenu suffisamment long pour permettre un défilement (scrolling). Si la page n'est pas plus grande que la fenêtre du navigateur, il n'y aura pas de défilement possible, et donc aucun événement scroll ne sera déclenché.
Si vous souhaitez tester ce code, assurez-vous d'ajouter suffisamment de contenu texte dans le corps de votre page index.html afin de permettre le défilement.

Événements sur les formulaires

Pour aborder les événements spécifiques aux formulaires, nous allons ajouter un formulaire dans la page index.html et définir son style dans la feuille de style my_style.css.

Les événements liés aux formulaires que nous allons explorer dans cette section incluent : submit, reset, focus, blur, change, et input.

Voici le code HTML du formulaire :

```
    <!DOCTYPE html>
    <html>

    <head>
        <meta charset="utf-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1">
        <title>Formation JavaScript</title>
                    <link rel="stylesheet" type="text/css"
href="my_style.css">
    </head>

    <body>
        <img id="mylogo" src="logo.png">
        <h1 id="title1">Formation JavaScript</h1>
        <div class="introduction">
            <h2 id="intro">Introduction au DOM</h2>
             <p>Voici le paragrpahe 1 avec du texte introductif
1</p>
             <p>Voici le paragrpahe 2 avec du texte introductif
2</p>
             <p>Voici le paragrpahe 3 avec du texte introductif
3</p>
        </div>
        <button class="btn">Cliquer ici</button>
        <br><br>
        <form id="myForm">
            <div id="resultat"></div>

            <label for="nom">Nom :</label>
            <input type="text" id="nom" name="nom"><br><br>

            <label for="email">Email :</label>
                            <input type="text" id="email"
name="email"><br><br>

                <label for="langages">Langages de programmation
:</label><br>
                        <input type="checkbox" id="langage1"
```

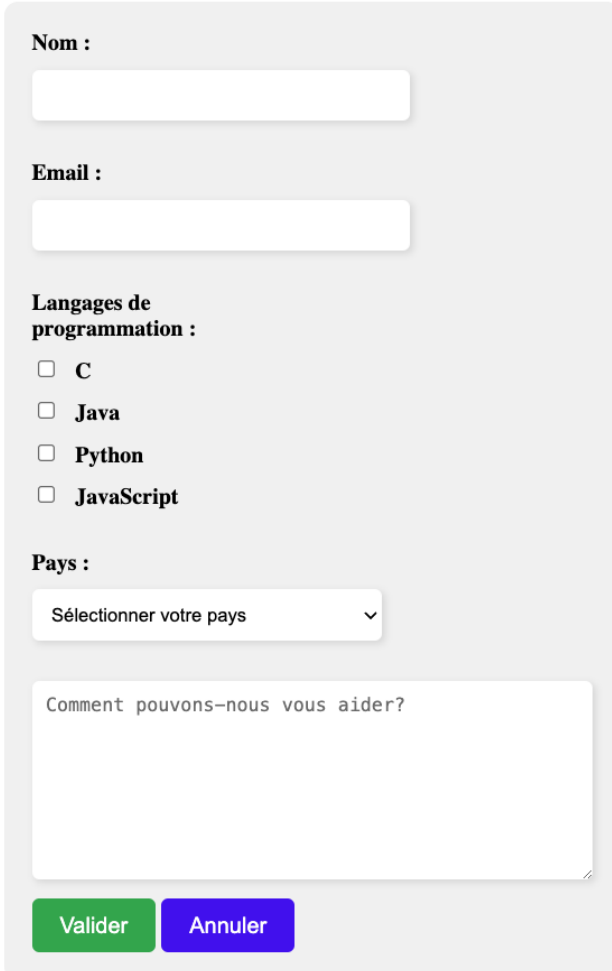
```
name="casesacocher" value="C">
            <label for="langage1">C</label><br>
                        <input  type="checkbox"  id="langage2"
name="casesacocher" value="Java">
            <label for="langage2">Java</label><br>
                        <input  type="checkbox"  id="langage3"
name="casesacocher" value="Python">
            <label for="langage3">Python</label><br>
                        <input  type="checkbox"  id="langage4"
name="casesacocher" value="JavaScript">
            <label for="langage4">JavaScript</label><br><br>

            <label for="pays">Pays :</label>
            <select id="pays" name="pays">
                    <option value="">Sélectionner votre pays</
option>
                <option value="france">France</option>
                <option value="espagne">Espagne</option>
                <option value="maroc">Maroc</option>
                <option value="algerie">Algérie</option>
                <option value="italie">Italie</option>
                <option value="allemagne">Allemagne</option>
            </select><br><br>

            <textarea name="" id="message" cols="30" rows="8"
                  placeholder="Comment pouvons-nous vous aider
?"></textarea><br><br>

                        <input  type="submit"  id="submit-btn"
value="Valider">
                          <input  type="reset"  id="reset-btn"
value="Annuler">
        </form>

        <script src="my_script.js"></script>
    </body>

    </html>
```

Voici le code CSS du style du formulaire:

```css
form {
    background-color: #f2f2f2;
    padding: 20px;
    width: 400px;
    border-radius: 10px;
    width: 400px;
    margin: auto;
}

label {
    display: inline-block;
    width: 150px;
    font-weight: bold;
    margin-bottom: 10px;
}

input[type="text"],
select {
    width: 250px;
    padding: 10px;
    margin-bottom: 10px;
    border: none;
    border-radius: 5px;
    box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.1);
}

input[type="checkbox"] {
    margin-right: 10px;
}

textarea {
    width: 380px;
    padding: 10px;
    margin-bottom: 10px;
    border: none;
    border-radius: 5px;
```

```
    box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.1);
}

input[type="submit"] {
    background-color: #4CAF50;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 16px;
}

input[type="submit"]:hover {
    background-color: #0bf054;
}

input[type="reset"] {
    background-color: #4e06f5;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 16px;
}

input[type="reset"]:hover {
    background-color: #f00b63;
}
```

Résultat :



L'événement input

L'événement input est déclenché à chaque modification dans un champ de formulaire, qu'il s'agisse d'une zone de texte ou d'un autre élément de saisie.

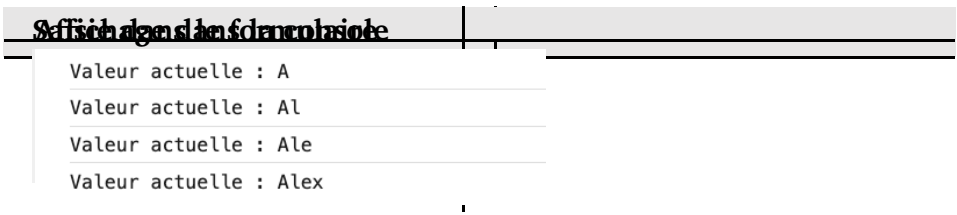
Le code JavaScript ci-dessous permet de capturer et d'afficher, en temps réel, la valeur saisie dans le champ de texte :

```
                                const        inputField        =
document.querySelector('input[type="text"]');
    inputField.addEventListener('input', function () {
        console.log("Valeur actuelle : " + inputField.value);
    });
```

Lorsque l'utilisateur commence à taper dans le champ de texte « Nom » par exemple, chaque caractère saisi sera affiché en temps réel dans la console grâce à l'événement input.

Ce code est utile pour des fonctionnalités telles que la validation en direct ou les suggestions en fonction des données saisies par l'utilisateur.

Résultat :



L'événement submit

L'événement submit est déclenché lorsque l'utilisateur soumet un formulaire. Il permet de valider les données ou de les traiter avant l'envoi.

Voici le code JavaScript, permettant de capturer l'événement de soumission du formulaire, tout en empêchant son envoi réel grâce à **event.preventDefault()**, et affiche une alerte indiquant que le formulaire a été soumis.

Voici le code JavaScript à étudier :

```
    const form = document.querySelector('form');
    form.addEventListener('submit', function (event) {
                event.preventDefault(); // Empêche l'envoi du
formulaire
        alert("Formulaire soumis !");
    });
```

Explication :

- **Le code JavaScript** : Lorsque l'utilisateur soumet le formulaire (clique sur le bouton « Valider »), l'événement **submit** est capturé. Grâce à **event.preventDefault()**, l'envoi du formulaire est empêché pour permettre

l'exécution d'autres actions, comme afficher une alerte.

- **Résultat** : Lorsque vous cliquez sur le bouton « Valider », une alerte avec le message "Formulaire soumis !" apparaîtra, mais le formulaire ne sera pas réellement envoyé au serveur.

Cela peut être très utile pour valider les données côté client avant l'envoi réel du formulaire, ou pour effectuer des actions telles que l'envoi des données via AJAX sans recharger la page.

L'événement reset

L'événement reset se déclenche lorsqu'un formulaire est réinitialisé. Cet événement peut être utilisé pour déclencher des actions spécifiques, comme afficher un message de confirmation de réinitialisation.

```
const resetBtn = document.getElementById('reset-btn');
resetBtn.addEventListener('click', function () {
    alert('Formulaire réinitialisé.');
});
```

Explication :

Le bouton de réinitialisation est associé à un événement **click** qui affiche un message confirmant que le formulaire a été réinitialisé.

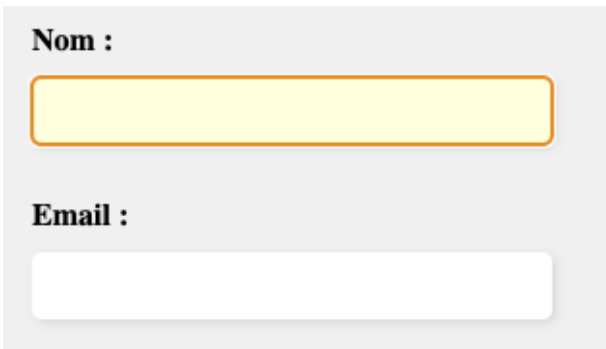
L'événement focus

L'événement focus est déclenché lorsqu'un élément (comme un champ de texte) reçoit le focus, c'est-à-dire que l'utilisateur clique ou navigue dans cet élément.

```
const nameInput = document.getElementById('nom');
nameInput.addEventListener('focus', function () {
    nameInput.style.backgroundColor = 'lightyellow';
});
```

Explication :

Lorsqu'on clique sur le champ Nom, l'événement focus se déclenche et change la couleur de fond du champ pour indiquer qu'il est en cours d'édition.



L'événement blur

L'événement **blur** se déclenche lorsque l'utilisateur quitte un champ de saisie. Il peut être utilisé pour valider les données lorsque l'utilisateur a fini de remplir le champ.

```
const nameInput = document.getElementById('nom');

nameInput.addEventListener('blur', function () {
    if (nameInput.value === '') {
        alert('Le champ Nom ne doit pas être vide.');
    }
});
```

Explication :

Dès que l'utilisateur **quitte** le champ Nom, le script vérifie si le champ est vide. Si c'est le cas, une alerte est déclenchée.

L'événement change

L'événement change se produit lorsque la valeur d'un élément de formulaire change (par exemple, un champ de sélection ou une case à cocher). Il est particulièrement utile pour les sélections et les cases à cocher.

```
    const countrySelect = document.getElementById('pays');
    countrySelect.addEventListener('change', function () {
                    alert('Vous avez sélectionné : ' +
countrySelect.value);
    });
```

Explication :

Lorsque l'utilisateur change le pays sélectionné dans le menu déroulant, une alerte affiche la nouvelle sélection.

L'événement input

L'événement input se déclenche chaque fois que l'utilisateur modifie le contenu d'un champ de texte. Cet événement est parfait pour surveiller les saisies en temps réel.

```
    const emailInput = document.getElementById('email');
    const resultDiv = document.getElementById('resultat');

    emailInput.addEventListener('input', function () {
```

```
        resultDiv.textContent = 'Adresse email saisie : ' +
emailInput.value;
    });
```

Explication :

- **const emailInput = document.getElementById('email');** : Cette ligne récupère l'élément HTML <input> avec l'ID email (qui correspond au champ de saisie de l'adresse email dans le formulaire). Cet élément est ensuite stocké dans la variable emailInput.
- **const resultDiv = document.getElementById('resultat');** : De la même manière, cette ligne récupère l'élément HTML avec l'ID resultat (qui est une div où l'on va afficher dynamiquement le texte lié à l'événement). Cet élément est ensuite stocké dans la variable resultDiv.

- **emailInput.addEventListener('input', ...)** : On attache ici un gestionnaire d'événement input au champ de saisie email. Cela signifie que chaque fois que l'utilisateur commence à taper ou modifie quelque chose dans ce champ, l'événement input sera déclenché.

L'événement input se déclenche chaque fois qu'une modification est faite dans un champ de saisie, qu'il s'agisse de l'ajout ou de la suppression de caractères.

- **function() {...}** : Cette fonction anonyme est déclenchée chaque fois que l'événement input se produit. Ce que cette fonction fait est simple : elle met à jour dynamiquement le contenu de la div **resultDiv** avec la valeur actuelle du champ de saisie **emailInput**.

- **resultDiv.textContent = 'Adresse email saisie : ' + emailInput.value;** : Cette ligne modifie le contenu texte de la div. Elle prend la chaîne de caractères 'Adresse email saisie : ' et y concatène la valeur actuellement saisie dans le champ email. La propriété **textContent** est utilisée ici pour définir le texte à l'intérieur de l'élément div.

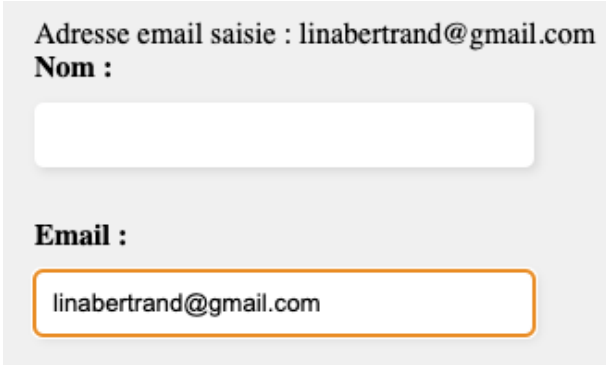
- **emailInput.value** : Cette propriété contient la valeur actuelle saisie dans le champ input. À chaque

modification, elle est mise à jour avec la nouvelle saisie de l'utilisateur.

Résultat :
Lorsque l'utilisateur commence à taper dans le champ email, le texte "Adresse email saisie : ", suivi de la valeur actuelle saisie, est mis à jour en temps réel dans la **div resultat**.

Par exemple :

Si l'utilisateur tape "linabertrand@gmail.com" dans le champ email, la **div resultat** affichera :



Utilité :
Cet exemple montre comment réagir en temps réel aux saisies des utilisateurs. Ce type de fonctionnalité est utile dans plusieurs cas pratiques :

- **Validation en direct** : Tu peux afficher un message d'erreur si l'email n'est pas au bon format.
- **Aide contextuelle** : Proposer des suggestions ou des retours immédiats à l'utilisateur.
- **Mise à jour dynamique** : Afficher les modifications saisies dans une autre partie de la page sans avoir à soumettre le formulaire.

Exemple d'amélioration avec une validation d'email :
Tu peux ajouter une validation simple pour vérifier si l'email contient un "@" et afficher un message d'erreur si ce n'est pas le cas :

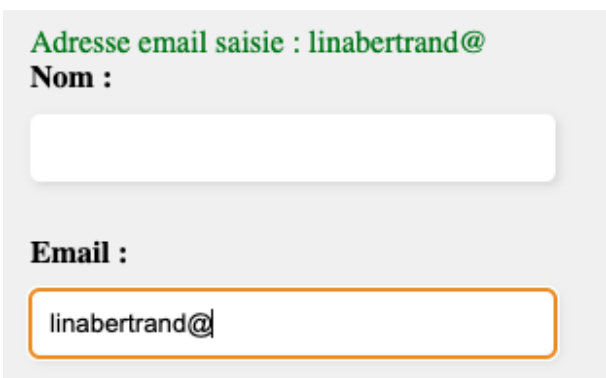
```
    emailInput.addEventListener('input', function () {
        if (emailInput.value.includes('@')) {
            resultDiv.textContent = 'Adresse email saisie : '
+ emailInput.value;
```

```
            resultDiv.style.color = 'green'; // Couleur verte
si l'email est correct
        } else {
            resultDiv.textContent = 'L\'adresse email n\'est
pas valide';
            resultDiv.style.color = 'red'; // Couleur rouge
pour indiquer une erreur
        }
    });
```

Avec ce code, si l'email ne contient pas le symbole "@", un message en rouge indiquera que l'adresse n'est pas valide.

Lorsque l'utilisateur commence à saisir du texte dans le champ email, un message "L'adresse email n'est pas valide" s'affiche en rouge tant que le symbole "@" n'a pas été saisi.

Dès que ce symbole est ajouté, l'adresse email complète saisie apparaît en vert dans la zone spécifiée.



Chapitre 17

Projet de développement d'une calculatrice

Dans ce chapitre, nous allons mettre en pratique nos connaissances en JavaScript en développant une calculatrice professionnelle. Ce projet va nous permettre d'explorer la manière dont JavaScript peut interagir avec le DOM pour manipuler des éléments HTML et gérer des événements. L'objectif ici est d'intégrer des fonctionnalités essentielles à une calculatrice telles que les opérations arithmétiques de base, la gestion des entrées utilisateur, et l'affichage des résultats.

Ce projet sera divisé en plusieurs étapes :

- La création de l'interface utilisateur (déjà faite avec le fichier HTML et la feuille de style CSS),
- L'ajout des fonctionnalités JavaScript pour permettre la manipulation des boutons et l'affichage des résultats.

Objectif principal

L'objectif de ce chapitre est l'élaboration du script JavaScript qui ajoutera les fonctionnalités interactives de la calculatrice. Nous allons voir comment capturer les clics sur les boutons, effectuer les calculs, gérer l'affichage des résultats et implémenter des fonctions comme la division, la multiplication, l'addition et la soustraction. Ce sera l'occasion d'appliquer les concepts fondamentaux du développement web et de la manipulation du DOM, tout en découvrant de nouvelles astuces de programmation.

Structure HTML et CSS de la calculatrice

Nous commençons par créer un dossier dédié à notre projet de calculatrice, que nous nommerons "calculator_V1". Tous les fichiers relatifs au développement de la calculatrice seront placés dans ce dossier.

Nous avons déjà mis en place une structure HTML de base dans le fichier calculator1.html. Ce fichier définit l'interface visuelle de la calculatrice, avec des boutons pour les chiffres, les opérateurs et un écran d'affichage pour montrer les résultats.

Le fichier style.css quant à lui, se charge de styliser la calculatrice, la rendant à la fois moderne et professionnelle. Nous utilisons des couleurs contrastées pour rendre les touches bien visibles, et des ombres pour donner un effet de relief à la calculatrice. Ce style CSS assure également la disposition en grille des boutons, facilitant la présentation des touches de la calculatrice.

Résumé du fichier HTML (calculator1.html)
Le fichier calculator1.html met en place l'interface utilisateur de la calculatrice. Il contient :

- Un champ d'affichage désactivé (l'élément `<input type="text" id="display">), utilisé pour montrer les résultats des calculs.
- Un ensemble de boutons représentant les chiffres, les opérateurs et des fonctions spéciales comme le bouton C (pour réinitialiser la calculatrice) et = (pour effectuer le calcul).
- Un lien vers la feuille de style **style.css** pour la mise en forme, et un lien pour intégrer les icônes à l'aide de FontAwesome (qui permet d'ajouter des icônes telles que le bouton de fermeture). (Pour utiliser les services du site FontAwesome, je vous invite consulter leur site web : https://fontawesome.com/).
- Nous avons également insérer un lien vers un fichier JavaScript scriptv1.js que nous allons créer dans la suite de ce chapitre.

Calculator.html :

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Calculatrice Professionnelle</title>
    <link rel="stylesheet" type="text/css" href="style.css">
       <script  src="https://kit.fontawesome.com/a62a90acc1.js"
crossorigin="anonymous"></script>
</head>

<body>
    <div class="calculator">
        <div class="calculator-header">
            <div class="calculator-icons">
                <i class="fa-solid fa-circle-xmark" id="close-
icon"></i>
            </div>
        </div>
        <input type="text" id="display" disabled>
        <div class="keypad">
            <button value="7">7</button>
            <button value="8">8</button>
            <button value="9">9</button>
              <button class="operator" id="btnDivide" value="/">÷</
button>
            <button value="4">4</button>
            <button value="5">5</button>
            <button value="6">6</button>
             <button class="operator" id="btnMultiply" value="*">×</
button>
            <button value="1">1</button>
            <button value="2">2</button>
            <button value="3">3</button>
               <button class="operator" id="btnMinus" value="-">-</
button>
                 <button class="decimal-number" id="decimal-number"
value=".">.</button>
            <button value="0">0</button>
             <button class="clear" id="btnClear" value="C">C</
button>
              <button class="operator" id="btnPlus" value="+">
```

```
+</button>
                <button class="btnEqual" id="btnEquals" value="=">=</
button>

        </div>
    </div>

    <script src="scriptv1.js"></script>
</body>

</html>
```

Résumé du fichier CSS (style.css)
Le fichier style.css est utilisé pour styliser l'apparence de la calculatrice. Voici quelques éléments clés :

- **Disposition et couleurs** : Un dégradé bleu est appliqué à l'arrière-plan de la page, tandis que la calculatrice utilise un thème foncé avec des boutons de couleur contrastée.
- **Grille de boutons** : Les boutons de la calculatrice sont organisés en grille, facilitant leur disposition avec une largeur uniforme pour chaque bouton.
- **Effets de survol** : Les boutons changent de couleur lorsqu'ils sont survolés, offrant un retour visuel à l'utilisateur.

```
    body {
        padding: 100px;
        background: #56CCF2;
        /* fallback for old browsers */
         background: -webkit-linear-gradient(to right, #2F80ED,
#56CCF2);
        /* Chrome 10-25, Safari 5.1-6 */
              background: linear-gradient(to right, #2F80ED,
#56CCF2);
          /* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera
12+, Safari 7+ */
```

```
}

.calculator {
    width: 320px;
    margin: 0 auto;
    padding: 5px 20px 20px;
    background-color: #212121;
    border-radius: 10px;
    box-shadow: 10px 20px 24px rgba(31, 26, 26, 0.7);
    box-sizing: border-box;
    /* Ajout de cette propriété */
    overflow: hidden;
    /* Ajout de cette propriété */
}

.calculator #display {
    width: 100%;
    margin-bottom: 20px;
    padding: 10px;
    font-size: 28px;
    text-align: right;
    background-color: #424242;
    color: #fff;
    border: none;
    border-radius: 5px;
    box-sizing: border-box;
    /* Ajout de cette propriété */
    overflow: hidden;
    /* Ajout de cette propriété */
}

.calculator .keypad {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    grid-gap: 10px;
}

.calculator .keypad button {
    width: 100%;
```

```
        padding: 20px;
        font-size: 20px;
        background-color: #757575;
        color: #fff;
        border: none;
        border-radius: 5px;
        box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);
        cursor: pointer;
    }

    .calculator .keypad button:hover {
        background-color: #616161;
    }

    .calculator .keypad button.operator {
        background-color: #ff9800;
    }

    .calculator .keypad button.operator:hover {
        background-color: #f57c00;
    }

    .calculator .keypad button.btnEqual {
        width: calc(400% + 30px);
          /* Définir la largeur du bouton "=" avec une valeur
calculée */
        background-color: #4a4a4a;
    }

    .calculator .keypad button.btnEqual:hover {
        background-color: #585757;
    }

    /*les boutons header*/
    .calculator .calculator-header {
        display: flex;
        justify-content: flex-start;
        align-items: flex-start;
        padding: 10px;
```

```
    margin-bottom: 20px;
}

.calculator .calculator-icons {
    display: flex;
    gap: 10px;
    cursor: pointer;
}

.calculator .calculator-icons i {
    cursor: pointer;
}

/*Couleurs des icones fermer et reduire */
#close-icon {
    color: #ff9800;
}
```

À ce stade, si vous ouvrez la page calculator1.html, vous devriez obtenir un résultat similaire à l'illustration suivante :

Objectifs de la suite

Dans les prochaines étapes, nous allons nous concentrer sur l'élaboration du fichier **scriptv1.js**, où nous allons définir les actions à effectuer lorsque l'utilisateur interagit avec les boutons. Nous allons également nous assurer que la calculatrice gère correctement les opérations et renvoie les résultats attendus.

Dans la prochaine partie, nous allons commencer à écrire le script JavaScript pour que la calculatrice devienne pleinement fonctionnelle. Soyez prêt à découvrir étape par étape, comment JavaScript peut donner vie à cette interface visuelle ! N'hésitez pas à tester votre code à chaque étape du développement.

Étape 1 : Mise en place du processus de fermeture de la calculatrice

Dans cette première étape, nous allons gérer la fonctionnalité permettant de fermer la calculatrice en cliquant sur l'icône de fermeture (l'élément close-icon).

```
let closeIcon = document.getElementById('close-icon');
```

Explication : Cette ligne récupère l'élément close-icon via son identifiant et le stocke dans la variable closeIcon.

Ensuite, nous ajoutons un écouteur d'événement pour détecter le clic sur cette icône et fermer la fenêtre du navigateur.

```
closeIcon.addEventListener('click', function () {
    window.close();
});
```

Explication : Lorsque l'utilisateur clique sur l'icône de fermeture, la méthode **window.close()** est appelée, fermant ainsi la fenêtre de la calculatrice.

Étape 2 : Gestion des boutons et affichage des valeurs

Nous commençons par récupérer les éléments HTML nécessaires pour la calculatrice, à savoir l'élément d'affichage et tous les boutons.

```
let display = document.getElementById("display");
let buttons = document.getElementsByTagName("button");
```

Explication :

- display est l'élément où les résultats des opérations et les entrées utilisateur seront affichés.
- buttons contient tous les boutons de la calculatrice (chiffres, opérateurs, etc.).
- Ensuite, nous utilisons l'événement **DOMContentLoaded** pour nous assurer que notre code est exécuté une fois que le DOM est complètement chargé.

```
    document.addEventListener("DOMContentLoaded", function ()
{
```

Explication : L'événement **DOMContentLoaded** est déclenché lorsque le document HTML est entièrement chargé et prêt à être manipulé. Cela garantit que les éléments HTML nécessaires sont bien présents avant que le script ne les manipule.

Ensuite, nous ajoutons un gestionnaire d'événements de clic pour chaque bouton, afin de récupérer et afficher la valeur du bouton cliqué.

```
    for (let i = 0; i < buttons.length; i++) {
        buttons[i].addEventListener("click", function () {
            let value = this.value;
            console.log(value);
        });
    }
```

Explication : Ce code est ajouté à l'intérieur des accolades de :

```
document.addEventListener("DOMContentLoaded", function (){
```

Chaque bouton récupère sa valeur lorsqu'il est cliqué et la valeur est affichée dans la console. Cela prouve que la récupération des valeurs fonctionne correctement.

Affichage des valeurs dans le champ d'affichage
Au lieu d'afficher les valeurs dans la console, nous les affichons directement dans le champ d'affichage de la calculatrice.

```
   display.value = value;
```

Explication : La valeur du bouton cliqué est directement affichée dans le champ display.
Cependant, pour conserver la valeur précédente à chaque clic (et éviter que la nouvelle valeur n'efface la précédente), nous utilisons l'opérateur + =.

```
display.value += value;
```

Gestion des calculs et de l'opérateur d'égalité
Lorsqu'un utilisateur clique sur "=", nous devons évaluer l'expression mathématique et afficher le résultat.

```
if (value === "=") {
    try {
        display.value = eval(display.value);
    } catch (error) {
        display.value = "Erreur";
    }
}
```

Explication : Si le bouton "=" est cliqué, la fonction **eval()** évalue l'expression mathématique. Si une erreur survient (par exemple, si l'expression est invalide), le message "Erreur" s'affiche à la place.

Ajout de la fonctionnalité du bouton "C" pour effacer l'affichage
Nous ajoutons maintenant la fonctionnalité du bouton "**C**" pour réinitialiser l'affichage.

```
else if (value === "C") {
    display.value = "";
}
```

Explication : Si le bouton "**C**" est cliqué, l'affichage est effacé en définissant **display.value** à une chaîne vide.

Gestion du point décimal et des opérateurs successifs

Pour éviter les erreurs liées à l'insertion répétée du point décimal ou d'opérateurs successifs, nous implémentons des vérifications supplémentaires.

Empêcher plusieurs points décimaux :

```
    if (value === "." && display.value.includes(".")) {
        return;
    }
```

Explication : Si un point existe déjà dans l'affichage, un nouveau point est refusé.
Empêcher les opérateurs successifs :

```
    let lastChar = display.value.slice(-1);
    if (isOperator(lastChar) && isOperator(value)) {
        return;
    }
```

Explication : Cette vérification empêche la répétition successive d'opérateurs comme "+", "-", etc.

Définition de la fonction isOperator :

```
function isOperator(character) {
    return character === "+" || character === "-" || character
=== "*" || character === "/";
}
```

Explication : Cette fonction retourne **true** si le caractère donné est un opérateur arithmétique.

Validation des entrées : chiffres, opérateurs, et décimaux
Nous ajoutons une condition pour valider que les entrées sont bien des chiffres, des opérateurs ou des points décimaux avant de les afficher.

```
  if (!isNaN(value) || isOperator(value) || value === ".")
```

Explication : Cette ligne vérifie si la valeur cliquée est un chiffre, un

opérateur ou un point décimal. Si c'est le cas, elle est autorisée dans l'affichage.

Voici le code complet à ce stade :

scriptv1.js :

```
    let closeIcon = document.getElementById('close-icon');
    // Récupérer les éléments de la calculatrice
      //l'élément display où s'affiche les opérations et le
résultat
    let display = document.getElementById("display");
    let buttons = document.getElementsByTagName("button");

     document.addEventListener("DOMContentLoaded", function ()
{

            // Ajouter un gestionnaire d'événement de clic à
chaque bouton
        for (let i = 0; i < buttons.length; i++) {
            buttons[i].addEventListener("click", function () {
                //Obtenir la valeur du bouton cliqué
                let value = this.value;

          // Vérifier si le bouton cliqué est un chiffre, un
opérateur ou le symbole décimal
                    if (!isNaN(value) || isOperator(value) ||
value === ".") {
          // Vérifier si le symbole décimal existe déjà dans
l'affichage
                                        if (value === "." &&
display.value.includes(".")) {
                            return; // Empêcher la répétition du
symbole décimal
                    }

   // Vérifier si le dernier caractère de l'affichage est un
opérateur
                    let lastChar = display.value.slice(-1);
                                if (isOperator(lastChar) &&
```

```
isOperator(value)) {
                            return; // Empêcher la répétition
successive des opérateurs
                    }

                     // Ajouter la valeur du bouton cliqué à
l'affichage
                    display.value += value;

                } else if (value === "=") {
                    try {
                     // Évaluer l'expression mathématique dans
l'affichage
                        display.value = eval(display.value);
                    } catch (error) {
                        // Gérer les erreurs d'évaluation
                        display.value = "Erreur";
                    }
                } else if (value === "C") {
                    // Effacer l'affichage
                    display.value = "";
                } else {
                      // Ajouter la valeur du bouton cliqué à
l'affichage
                    display.value += value;
                }
            });
        }

    });

     // Vérifier si le caractère donné est un opérateur (+, -,
*, /)
    function isOperator(character) {
          return character === "+" || character === "-" ||
character === "*" || character === "/";
    }
    //Fermer la calculatrice
    closeIcon.addEventListener('click', function () {
```

```
        window.close();
    });
```

À ce stade, la calculatrice est pleinement fonctionnelle pour les opérations de base, la gestion des points décimaux, et la prévention des erreurs liées à l'utilisation incorrecte des opérateurs. De plus, la fonction de réinitialisation avec le bouton "C" est bien implémentée. Il reste toujours possible d'ajouter des fonctionnalités supplémentaires, mais le cœur du projet est en place et fonctionne bien.

Je te conseille de tester le code complet à ce stade et de l'améliorer si nécessaire !

NB. Vous pouvez télécharger le fichier **scriptv1.js** à partir du lien fourni à la fin de ce livre.

Dans la suite nous allons améliorer cette première version

Améliorations de la première version de la calculatrice

La version actuelle de la calculatrice fonctionne globalement bien, mais des tests ont révélé plusieurs points à améliorer :

- **Affichage constant** : Actuellement, tout ce que l'utilisateur tape (chiffres, opérateurs) s'affiche à l'écran de la calculatrice.
- **Problème de réinitialisation** : Lorsque le résultat d'un calcul est affiché, si l'utilisateur souhaite effectuer une nouvelle opération sans réinitialiser l'écran, les nouveaux chiffres ou opérateurs sont simplement ajoutés au résultat précédent, au lieu de commencer un nouveau calcul. En d'autres termes, avant d'effectuer une nouvelle opération, l'utilisateur doit d'abord réinitialiser manuellement l'écran.

Pour corriger ces dysfonctionnements, nous allons adopter une approche différente en évitant d'utiliser la méthode **eval()** qui était employée dans la version précédente.

Nous allons commencer par créer un nouveau fichier nommé **scriptv2.js** dans lequel nous allons transférer tout le code de **scriptv1.js** afin d'y apporter des améliorations. N'oubliez pas d'ajouter le lien vers ce fichier dans votre page calculator.html pour que la calculatrice fonctionne correctement avec le nouveau script.
Remarque : Vous pourrez télécharger ce fichier complet en utilisant le lien disponible à la fin de ce livre.

Étape 1 : Récupérer les valeurs des boutons et les afficher correctement à l'écran

Nous allons commencer par récupérer les valeurs des boutons et les afficher à l'écran, en assurant un meilleur contrôle sur les saisies de l'utilisateur.

```
    // Récupérer les éléments de la calculatrice
    let display = document.getElementById("display");
    let buttons = document.getElementsByTagName("button");
    document.addEventListener("DOMContentLoaded", function ()
{
        let currentValue = ""; // Valeur en cours d'écriture
            // Ajouter un gestionnaire d'événement de clic à
```

```
chaque bouton
        for (let i = 0; i < buttons.length; i++) {
            buttons[i].addEventListener("click", function () {
                // Obtenir la valeur du bouton cliqué
                let value = this.value;
                    // Vérifier si le bouton cliqué est un
chiffre, un opérateur ou le symbole décimal
                if (!isNaN(value) || value === ".") {
                    // Vérifier si le symbole décimal existe
déjà dans la valeur en cours d'écriture
                                        if (value === "." &&
currentValue.includes(".")) {
                            return; // Empêcher la répétition du
symbole décimal
                    }
                     // Ajouter la valeur du bouton cliqué à la
valeur en cours d'écriture
                    currentValue += value;
                    //console.log(currentValue);
                    display.value = currentValue;
                }
            });
        }
    });
```

Explication détaillée du code :

Récupération des éléments de la calculatrice :

```
let display = document.getElementById("display");
let buttons = document.getElementsByTagName("button");
```

Ces deux lignes permettent de récupérer les éléments essentiels de la calculatrice à partir du document HTML :

- **display** : Il s'agit de l'élément d'affichage de la calculatrice, qui est un champ de texte désactivé (input), identifié par l'ID "display". Cet élément affiche le résultat

en cours ou les valeurs saisies par l'utilisateur.

- **buttons** : Cette variable contient une collection de tous les boutons de la calculatrice. En utilisant la méthode **getElementsByTagName("button"),** on récupère tous les éléments HTML de type button présents dans le document, renvoyés sous forme de liste.

Gestionnaire d'événements DOMContentLoaded :

```
document.addEventListener(« DOMContentLoaded », function () {
```

Cette ligne ajoute un gestionnaire d'événement qui s'exécute lorsque le document HTML a été complètement chargé. Cela garantit que le script s'exécute uniquement après le chargement complet de la page, afin d'éviter des erreurs liées à des éléments encore non disponibles dans le DOM.

Initialisation de la variable currentValue :

```
let currentValue = ""; // Valeur en cours d'écriture
```

La variable **currentValue** est utilisée pour stocker la chaîne de caractères représentant la valeur en cours de saisie par l'utilisateur. Elle est initialement vide, ce qui signifie qu'aucune valeur n'a encore été saisie.

Parcours des boutons et ajout des gestionnaires d'événements :

```
for (let i = 0; i < buttons.length; i++) {
    buttons[i].addEventListener("click", function () {
        // Obtenir la valeur du bouton cliqué
        let value = this.value;
```

Cette boucle for parcourt tous les boutons de la calculatrice. À chaque itération, elle ajoute un gestionnaire d'événement pour chaque bouton, déclenché lors d'un clic. Lorsqu'un bouton est cliqué, la fonction anonyme s'exécute. La valeur du bouton cliqué est obtenue avec **this.value**, et est stockée dans la variable value.

Vérification de la validité de la saisie :

```
  if (!isNaN(value) || value === ".") {
```

Cette condition permet de vérifier si la valeur du bouton cliqué est un chiffre (un nombre) ou le symbole décimal (**.**"). La fonction **isNaN(value)** vérifie si value est "Not-a-Number" (c'est-à-dire si ce n'est pas un nombre). Si cette vérification retourne false, cela signifie que la valeur est un chiffre. Le symbole décimal "." est également accepté. Si la condition est vraie, l'exécution continue ; sinon, le bouton ne sera pas pris en compte.

Empêcher la répétition du symbole décimal :

```
if (value === "." && currentValue.includes(".")) {
    return; // Empêcher la répétition du symbole décimal
}
```

Cette condition empêche l'utilisateur d'ajouter plusieurs symboles décimaux dans une même entrée. Si l'utilisateur clique sur le point décimal (".") et que **currentValue** contient déjà un point, l'exécution de la fonction s'arrête avec return, empêchant ainsi la répétition du symbole décimal dans la chaîne de caractères.

Mise à jour de la valeur affichée :

```
currentValue += value;
display.value = currentValue;
```

Si les conditions précédentes sont validées, la valeur du bouton cliqué est ajoutée à la variable **currentValue** à l'aide de l'opérateur de concaténation (+=). Cela permet d'ajouter chaque chiffre ou symbole décimal saisi par l'utilisateur à la chaîne en cours. Ensuite, la valeur mise à jour de **currentValue** est affichée dans le champ d'affichage display, grâce à l'attribut value de cet élément. Cela met à jour dynamiquement l'affichage à chaque saisie.

En résumé, ce code de cette étape 1, permet à l'utilisateur de saisir des chiffres et un symbole décimal dans la calculatrice. Les valeurs sont affichées en temps réel à l'écran. Le contrôle sur la saisie garantit qu'aucun point décimal supplémentaire n'est ajouté, assurant ainsi une

saisie correcte des nombres décimaux.

Étape 2 : Amélioration du code pour gérer le premier opérateur et la première valeur

Dans cette étape, nous allons améliorer la gestion de la saisie de l'utilisateur pour détecter lorsque le premier opérateur est sélectionné et enregistrer la première valeur avant de permettre la saisie de la seconde valeur. Pour ce faire, nous introduisons deux nouvelles variables :

```
let firstValue = null; // Stocke la première valeur saisie
let operator = null;   // Stocke l'opérateur choisi
```

Ces variables nous permettront de conserver les informations nécessaires lors de l'opération mathématique.

Ajout d'une condition pour gérer l'opérateur :
Ensuite, nous modifions le bloc if existant (qui vérifie si la valeur du bouton est un chiffre ou un point décimal) pour y inclure une nouvelle condition else if, qui vérifie si l'utilisateur a sélectionné un opérateur :

```
else if (isOperator(value)) {
    if (firstValue === null) {
            // Si c'est le premier opérateur, enregistrer la
première valeur
          firstValue = parseFloat(currentValue);   // Convertir
la valeur en nombre flottant
        operator = value;  // Enregistrer l'opérateur choisi
         currentValue = ""; // Réinitialiser la valeur en cours
d'écriture
        // Affichage de tests intermédiaires pour le débogage
        console.log("Première valeur:", firstValue);
        console.log("Opérateur:", operator);
    }
}
```

Voici ce que fait cette partie du code :

- **Vérification de l'opérateur :** La fonction isOperator() est utilisée pour vérifier si la valeur du bouton cliqué est

un opérateur (+, -, *, /).

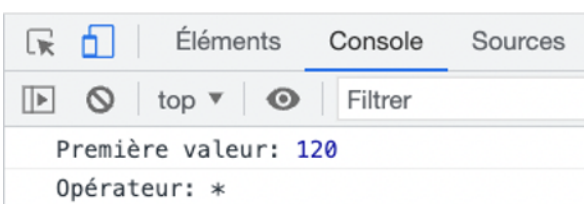
- **Enregistrement de la première valeur et de l'opérateur :** Si la valeur est un opérateur, cela signifie que l'utilisateur a choisi d'effectuer une opération. Nous devons alors :
    - Enregistrer la première valeur (currentValue) en utilisant parseFloat() pour la convertir en nombre flottant.
    - Stocker l'opérateur choisi dans la variable operator.
    - Réinitialiser la variable currentValue pour préparer la saisie de la deuxième valeur.
- **Test intermédiaire** : Les instructions console.log() servent à vérifier que la première valeur et l'opérateur sont correctement enregistrés. Ces lignes seront supprimées dans la version finale du script.

Explication de la condition if (firstValue === null) :

La condition **if (firstValue === null)** vérifie **si aucune première valeur n'a encore été enregistrée**. En effet, firstValue est initialisée avec null au début pour indiquer qu'aucune donnée n'a été saisie. Lorsque l'utilisateur sélectionne un opérateur pour la première fois, nous savons alors qu'il a entré une première valeur, et cette condition nous permet de capturer cette valeur.

Si firstValue est toujours null au moment du clic sur un opérateur, cela signifie que nous sommes dans la phase où la première valeur doit être enregistrée. Si firstValue n'est pas null, cela indique que la première valeur a déjà été enregistrée, et le code peut alors continuer avec l'opération de calcul.

Test de la saisie du premier nombre et choix de l'opérateur :



Définition de la fonction isOperator :

```
// Fonction pour vérifier si un caractère est un opérateur (+, -, *, /)
function isOperator(character) {
    return character === "+" || character === "-" || character === "*" || character === "/";
}
```

La fonction isOperator() permet de vérifier si un caractère donné correspond à l'un des opérateurs mathématiques standards (+, -, *, /). Cette fonction est utilisée dans la condition else if pour vérifier si le bouton cliqué correspond à un opérateur.

En résumé, cette étape permet de détecter lorsque l'utilisateur choisit un opérateur pour la première fois. Le code enregistre alors la première valeur saisie ainsi que l'opérateur sélectionné, avant de permettre à l'utilisateur de saisir la deuxième valeur. Grâce à ces ajustements, la calculatrice est maintenant capable de différencier la saisie de la première valeur de celle de la seconde, tout en offrant une interface plus claire et fluide pour l'utilisateur.

Étape 3 : Le cas où la première valeur est déjà enregistrée
À cette étape, nous allons gérer le cas où une première valeur (firstValue) a déjà été enregistrée. Nous devons maintenant effectuer un calcul intermédiaire chaque fois qu'un nouvel opérateur est cliqué après la saisie d'une deuxième valeur.

```
    else if (isOperator(value)) {
        if (firstValue === null) {
            // Si c'est le premier opérateur, enregistrer la
première valeur
            firstValue = parseFloat(currentValue);
            operator = value;
              currentValue = ""; // Réinitialiser la valeur en
cours d'écriture
            // Test intermédiaire
            console.log("Première valeur:", firstValue);
            console.log("Opérateur:", operator);
        } else {
                // Si un opérateur existe déjà, effectuer le
calcul
            let secondValue = parseFloat(currentValue);
                    let result = calculateResult(firstValue,
secondValue, operator);
                // Mettre à jour les valeurs pour le calcul
suivant
            firstValue = result;
            operator = value;
              currentValue = ""; // Réinitialiser la valeur en
cours d'écriture
            display.value = result;
            // Tests intermédiaires
            console.log("Première valeur:", firstValue);
            console.log("Opérateur:", operator);
            console.log("Deuxième valeur:", secondValue);
            console.log("Résultat intermédiaire:", result);
        }
    }
```

Explications détaillées :
Ce bloc de code gère deux cas :

1. **Enregistrement de la première valeur et de l'opérateur** : Si **firstValue** est encore null (c'est-à-dire que l'utilisateur vient juste de saisir une première valeur et clique pour la première fois sur un opérateur), le

programme :

- Convertit la **currentValue** (valeur en cours de saisie) en un nombre à l'aide de **parseFloat().**
- Stocke cette valeur dans **firstValue**.
- Enregistre l'opérateur choisi dans la variable **operator**.
- Réinitialise la **currentValue** pour permettre la saisie de la deuxième valeur.
- Les messages de console.log permettent de vérifier que la première valeur et l'opérateur sont correctement enregistrés.

1. **Calcul intermédiaire** : Si **firstValue** n'est plus null, cela signifie qu'une première valeur a déjà été saisie avec un opérateur. Dans ce cas, l'utilisateur est en train de saisir la deuxième valeur. Lorsque l'utilisateur clique sur un nouvel opérateur ou sur =, le programme :

- Convertit la **currentValue** en nombre et la stocke dans **secondValue**.
- Utilise la fonction **calculateResult()** pour calculer le résultat en fonction de l'opérateur et des deux valeurs (**firstValue** et **secondValue**).
- Met à jour **firstValue** avec le résultat du calcul intermédiaire.
- Affiche ce résultat dans le champ d'affichage **display.value**.
- Réinitialise la **currentValue** pour permettre la saisie suivante.

Les messages de console.log sont utilisés pour afficher les valeurs à chaque étape et vérifier que le calcul intermédiaire est bien exécuté. Ils seront supprimés dans la version finale.

Fonction calculateResult() :

```
      // Fonction pour calculer le résultat de l'expression
mathématique
          function calculateResult(firstValue, secondValue,
operator) {
        switch (operator) {
            case "+":
                return firstValue + secondValue;
            case "-":
```

```
                return firstValue - secondValue;
            case "*":
                return firstValue * secondValue;
            case "/":
                return firstValue / secondValue;
            default:
                return null;
        }
    }
```

Explication de la fonction calculateResult() :

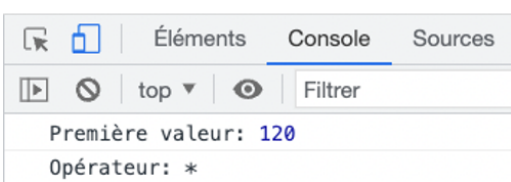
Cette fonction effectue les calculs mathématiques en fonction de l'opérateur choisi :

- Elle prend en paramètres **firstValue**, **secondValue** et **operator**.
- En fonction de l'opérateur (+, -, *, /), elle effectue l'opération correspondante sur les deux valeurs et renvoie le résultat.
- Si l'opérateur n'est pas reconnu, la fonction renvoie null.

Déroulement des tests intermédiaires :

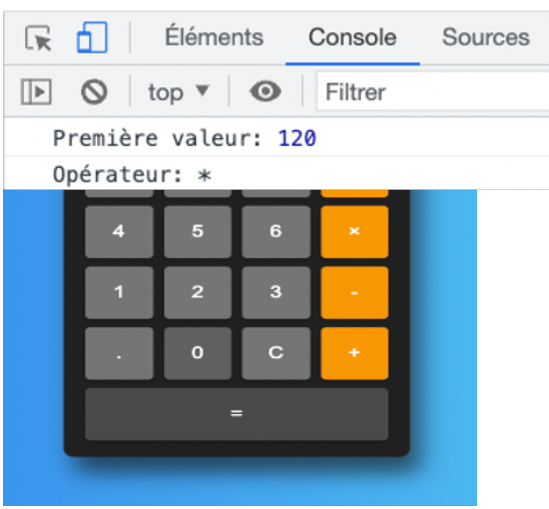
Action 1 : Saisie du premier nombre et choix de l'opérateur :

Lorsque l'utilisateur saisit un premier nombre puis clique sur un opérateur, ce nombre est enregistré dans **firstValue** et l'opérateur dans **operator**.



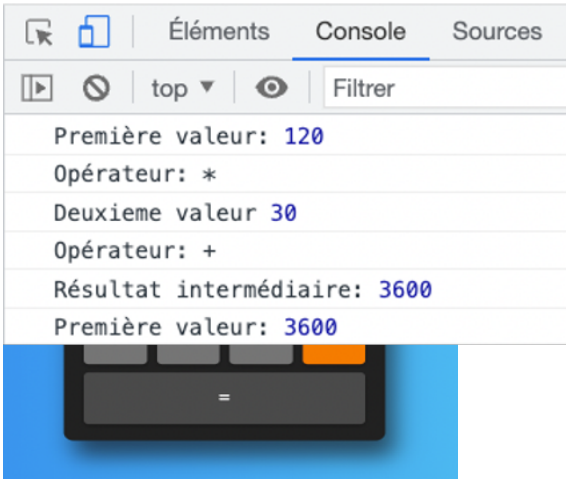
Action 2 : Saisie du deuxième nombre :

Après avoir cliqué sur l'opérateur, l'utilisateur saisit un deuxième nombre qui est stocké dans **currentValue**. La console reste inchangée à cette étape.



Action 3 : Clic sur un nouvel opérateur :

Lorsque l'utilisateur clique sur un autre opérateur, le calcul intermédiaire est effectué et le résultat est affiché. Ce résultat devient la nouvelle **firstValue** pour le prochain calcul. Les messages affichés dans la console montrent bien le déroulement du calcul intermédiaire.



En résume, dans cette étape nous avons ajouté du code qui permet de gérer les calculs intermédiaires en stockant les valeurs saisies et en effectuant des calculs à chaque fois qu'un nouvel opérateur est cliqué. Cela garantit que l'utilisateur peut enchaîner plusieurs opérations sans devoir réinitialiser la calculatrice après chaque résultat.

Le bouton d'égalité n'est pas encore fonctionnel dans cette étape, mais il sera ajouté dans une phase ultérieure pour gérer le calcul final.

Étape 4 : La fonction du bouton d'égalité
À ce stade du développement, nous avons déjà traité les cas où l'utilisateur clique sur un chiffre, un opérateur ou le symbole décimal. Désormais, nous allons nous concentrer sur la gestion du bouton d'égalité (=), qui permet d'afficher le résultat final du calcul.

Nous ajoutons une condition **else if** pour traiter spécifiquement ce cas :

```
    else if (value === "=") {
        // Si on a une valeur et un opérateur
        if (firstValue !== null && operator !== null) {
            let secondValue = parseFloat(currentValue);
                    let result = calculateResult(firstValue,
secondValue, operator);
            // Afficher le résultat final
            display.value = result;
            // Réinitialiser les valeurs
            firstValue = null;
```

```
            operator = null;
            currentValue = "";
            // Test intermédiaire
            console.log("Première valeur:", firstValue);
            console.log("Opérateur:", operator);
            console.log("Valeur actuelle:", currentValue);
            console.log("Résultat final:", result);
        }
    }
```

Explication détaillée de ce bloc de code :

- **Vérification des conditions pour effectuer le calcul final**

Le bloc de code commence par vérifier si les variables firstValue et operator sont différentes de null. Cette vérification est importante, car elle garantit que l'utilisateur a bien saisi une première valeur et a sélectionné un opérateur avant de demander le calcul final. Si ces conditions sont remplies, le calcul peut être exécuté.

- **Conversion de la deuxième valeur**

La deuxième valeur saisie par l'utilisateur est stockée dans currentValue. Avant de l'utiliser pour le calcul, nous la convertissons en nombre flottant grâce à parseFloat(), puis nous l'attribuons à la variable secondValue.

- **Calcul du résultat**

Une fois les deux opérandes (firstValue et secondValue) et l'opérateur obtenus, nous utilisons la fonction calculateResult() pour effectuer le calcul approprié (addition, soustraction, multiplication ou division) en fonction de l'opérateur sélectionné. Le résultat de ce calcul est stocké dans la variable result.

- **Affichage du résultat**

Le résultat obtenu est ensuite affiché dans l'écran de la calculatrice en assignant la valeur de result à display.value, ce qui permet à l'utilisateur de voir le résultat de l'opération.

- **Réinitialisation des valeurs**

Après le calcul, les variables firstValue, operator, et currentValue sont réinitialisées :

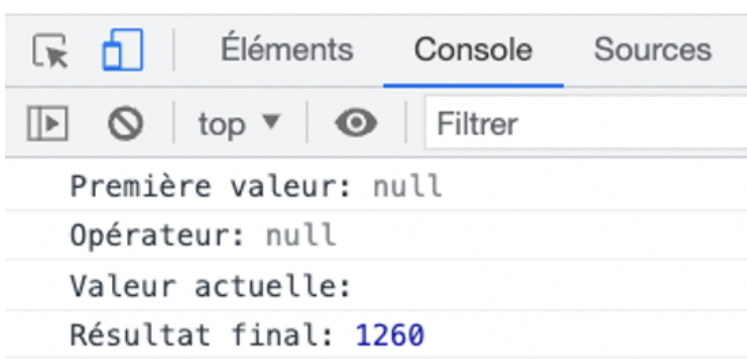
firstValue et operator sont remis à null, indiquant que le calcul est

terminé.
currentValue est réinitialisée à une chaîne vide, ce qui permet de commencer un nouveau calcul sans interférence avec les valeurs précédentes.

- **Tests intermédiaires**

Des messages de test intermédiaires sont affichés dans la console pour vérifier le bon déroulement du calcul. Ces messages montrent que :

- firstValue est bien réinitialisée à null.
- operator est également remis à null.
- currentValue est vide.
- Le résultat final du calcul est correctement calculé et affiché dans result.



En résumé :
Ce bloc de code permet de gérer le calcul final lorsque l'utilisateur clique sur le bouton = de la calculatrice. Il s'assure que les valeurs nécessaires au calcul sont présentes, effectue l'opération, affiche le résultat et réinitialise les variables pour préparer la calculatrice à un nouveau calcul.

Grâce à cette gestion, la calculatrice est désormais capable d'effectuer une série de calculs et d'afficher correctement le résultat final sur son écran.

Étape 5 : Le bouton de réinitialisation
Jusqu'à présent, nous avons mis en place les fonctionnalités de tous les boutons sauf celui de réinitialisation. Ce bouton permet de réinitialiser toutes les valeurs et l'affichage de la calculatrice pour permettre à l'utilisateur de commencer un nouveau calcul. Voici comment nous procédons pour ajouter cette fonctionnalité.

Nous ajoutons un nouveau bloc **else if** dans le gestionnaire

d'événements, qui vérifie si la valeur du bouton cliqué est "C". Si c'est le cas, nous réinitialisons l'affichage et toutes les valeurs de la calculatrice.

```
else if (value === "C") {
    // Effacer l'affichage et réinitialiser les valeurs
    display.value = "";
    currentValue = "";
    firstValue = null;
    operator = null;
}
```

Explication du code :
Ce bloc de code s'exécute lorsque l'utilisateur clique sur le bouton "C" (qui représente la fonction de réinitialisation). Il permet de réinitialiser toutes les valeurs et l'affichage de la calculatrice, comme suit :

- **display.value** = "" : Cela vide l'affichage de la calculatrice en le remettant à une chaîne vide. Cela signifie que tout ce qui était affiché précédemment est effacé.
- **currentValue** = "" : Cela réinitialise la valeur en cours de saisie (qui correspond à la saisie actuelle de l'utilisateur) en la remettant à une chaîne vide.
- **firstValue** = **null** : Cela réinitialise la première valeur enregistrée pour le calcul, indiquant qu'aucune valeur n'a été mémorisée.
- **operator** = **null** : Cela réinitialise l'opérateur sélectionné pour le calcul. Aucune opération n'est donc en attente d'exécution après la réinitialisation.

En résumé :
Lorsque l'utilisateur appuie sur le bouton "C", ce code vide l'affichage, réinitialise toutes les valeurs et remet la calculatrice dans son état initial, prête pour un nouveau calcul.

Voici la version finale de la version 2 de notre script scripv2.js qui prend en charge toutes les fonctions, y compris la réinitialisation :

```
    let closeIcon = document.getElementById('close-icon');
    // Récupérer les éléments de la calculatrice
    let display = document.getElementById("display");
    let buttons = document.getElementsByTagName("button");

    document.addEventListener("DOMContentLoaded", function ()
{

        let currentValue = ""; // Valeur en cours d'écriture
              let  firstValue  =  null;  //Stocke  les  valeurs
enregistrées
        let operator = null; //Stocke l'opérateur choisi
           // Ajouter un gestionnaire d'événement de clic à
chaque bouton
        for (let i = 0; i < buttons.length; i++) {
            buttons[i].addEventListener("click", function () {
                // Obtenir la valeur du bouton cliqué
                let value = this.value;
                      // Vérifier si le bouton cliqué est un
chiffre, un opérateur ou le symbole décimal
                if (!isNaN(value) || value === ".") {
                      // Vérifier si le symbole décimal existe
déjà dans la valeur en cours d'écriture
                                          if  (value  ===  "."  &&
currentValue.includes(".")) {
                            return; // Empêcher la répétition du
symbole décimal
                    }

  // Ajouter la valeur du bouton cliqué à la valeur en cours
d'écriture
                    currentValue += value;
                    //console.log(currentValue);
                    display.value = currentValue;
                } else if (isOperator(value)) {
                    if (firstValue === null) {
     // Si c'est le premier opérateur, enregistrer la première
valeur
                        firstValue = parseFloat(currentValue);
```

```
                        operator = value;
                         currentValue = ""; // Réinitialiser la
valeur en cours d'écriture
                        // Test intermédiaire
                              //console.log("Première valeur:",
firstValue);
                        //console.log("Opérateur:", operator);

                    } else {
                   // Si un opérateur existe déjà, effectuer le
calcul
                                                let  secondValue  =
parseFloat(currentValue);
                                                    let  result  =
calculateResult(firstValue, secondValue, operator);

                   // Mettre à jour les valeurs pour le calcul
suivant
                        firstValue = result;
                        operator = value;
                         currentValue = ""; // Réinitialiser la
valeur en cours d'écriture
                        display.value = result;
                        // Test intermédiaire
                                console.log("Deuxieme  valeur",
secondValue);
                        console.log("Opérateur:", operator);
                         console.log("Résultat intermédiaire:",
result);
                                 console.log("Première  valeur:",
firstValue);
                    }
                } else if (value === "=") {
                    //Si on a une valeur et un opérateur
                       if (firstValue !== null && operator !==
null) {
                                                let  secondValue  =
parseFloat(currentValue);
                                                    let  result  =
```

```
calculateResult(firstValue, secondValue, operator);
                        // Afficher le résultat final
                        display.value = result;
                        // Réinitialiser les valeurs
                        firstValue = null;
                        operator = null;
                        currentValue = "";

                        // Test intermédiaire
                                console.log("Première valeur:",
firstValue);
                        console.log("Opérateur:", operator);
                                console.log("Valeur actuelle:",
currentValue);
                                 console.log("Résultat final:",
result);
                    }
                } else if (value === "C") {
            // Effacer l'affichage et réinitialiser les valeurs
                    display.value = "";
                    currentValue = "";
                    firstValue = null;
                    operator = null;
                }
            });
        }
    });

     // Vérifier si le caractère donné est un opérateur (+, -,
*, /)
    function isOperator(character) {
           return character === "+" || character === "-" ||
character === "*" || character === "/";
    }

      // Fonction pour calculer le résultat de l'expression
mathématique
          function  calculateResult(firstValue,  secondValue,
operator) {
```

```
        switch (operator) {
            case "+":
                return firstValue + secondValue;
            case "-":
                return firstValue - secondValue;
            case "*":
                return firstValue * secondValue;
            case "/":
                return firstValue / secondValue;
            default:
                return null;
        }
    }

    //Fermer la calculatrice
    closeIcon.addEventListener('click', function () {
        window.close();
    });
```

Ce code couvre toutes les fonctionnalités essentielles de la calculatrice : saisie des chiffres, des opérateurs, calcul des résultats, gestion des nombres décimaux et réinitialisation. Vous pouvez désormais tester cette version pour vérifier que toutes les opérations fonctionnent correctement, y compris la réinitialisation avec le bouton "C".

Conclusion

Dans ce projet, nous avons construit une calculatrice fonctionnelle en utilisant HTML, CSS, et JavaScript. À travers ce développement, nous avons appliqué des concepts fondamentaux de la programmation, tels que la manipulation du DOM, l'écoute d'événements et la gestion des données en JavaScript. En partant d'une interface simple, nous avons ajouté des fonctionnalités essentielles comme les opérations arithmétiques de base, la gestion des nombres décimaux, et la réinitialisation des valeurs.

Nous avons également amélioré la structure du code en évitant l'usage de méthodes telles que eval() pour une meilleure sécurité et maintenabilité. Les tests réalisés tout au long du projet nous ont permis de valider chaque étape du développement, assurant une expérience utilisateur fluide.

Ce projet vous a permis de comprendre comment gérer les interactions utilisateur dans une interface web, mais aussi de mettre en place des techniques d'encapsulation, de validation et de modularité dans le code. Vous pouvez désormais personnaliser et étendre ce projet pour ajouter de nouvelles fonctionnalités ou explorer d'autres concepts, comme l'intégration d'opérations plus complexes ou une gestion avancée des erreurs.

En somme, ce projet constitue une base solide pour approfondir vos compétences en développement front-end et manipulation du DOM avec JavaScript.

Chapitre 18

La programmation asynchrone en JavaScript

La programmation asynchrone est l'une des caractéristiques les plus importantes et puissantes de JavaScript. Elle permet au langage de gérer des opérations qui prennent du temps, comme les appels réseau, les interactions avec les bases de données, ou la gestion des événements utilisateur, sans bloquer l'exécution du programme. En d'autres termes, la programmation asynchrone permet à JavaScript de continuer à exécuter d'autres tâches pendant qu'il attend la fin d'une opération longue.

JavaScript, par nature, est un langage mono-thread, ce qui signifie qu'il n'exécute qu'une tâche à la fois. Cependant, grâce à la programmation asynchrone, JavaScript peut simuler un comportement multitâche en utilisant des mécanismes tels que les promesses, les callbacks, et plus récemment, les fonctions async/await. Ces outils permettent aux développeurs de gérer des tâches complexes et de garantir que l'expérience utilisateur reste fluide, même lorsque des opérations lentes sont en cours.

Dans ce chapitre, nous allons plonger dans les principes fondamentaux de la programmation asynchrone. Nous commencerons par les callbacks, puis explorerons les promesses, avant de nous concentrer sur la syntaxe moderne async/await, qui simplifie grandement la gestion des opérations asynchrones. À la fin de ce chapitre, vous serez capable de gérer efficacement les opérations asynchrones dans vos projets JavaScript, qu'il s'agisse de récupérer des données à partir d'une API, de lire des fichiers ou de répondre à des événements utilisateur complexes.

Êtes-vous prêt à maîtriser la programmation asynchrone en JavaScript ? Allons-y !

Modes de fonctionnement en JavaScript : Synchrone et Asynchrone

En JavaScript, il existe deux modes principaux d'exécution des tâches : le mode synchrone et le mode asynchrone. Chacun de ces modes détermine la manière dont les tâches sont exécutées dans le script.

JavaScript synchrone :
Le mode synchrone en JavaScript signifie que chaque tâche doit être terminée avant qu'une autre puisse commencer. Cela implique que le code est exécuté de manière séquentielle et bloquante. Autrement dit, si une tâche prend du temps, toutes les autres tâches qui suivent devront attendre que celle-ci soit complètement terminée avant de pouvoir démarrer. Le programme est donc exécuté ligne après ligne, et chaque ligne doit avoir terminé son exécution avant que la suivante puisse commencer.

Par exemple :

```
console.log("A");
console.log("B");
console.log("C");
console.log("D");
console.log("E");
console.log("F");
console.log("G");
console.log("H");
```

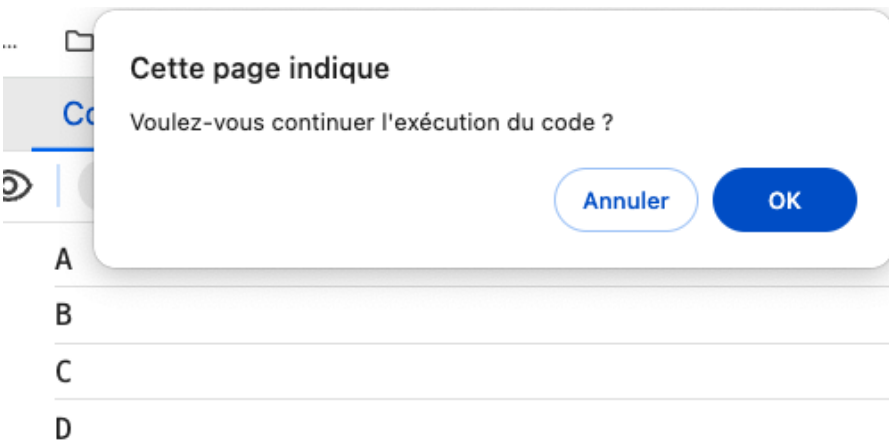
Dans ce cas, les messages seront affichés dans l'ordre, un après l'autre, sans interruption.

```
A
B
C
D
E
F
G
H
```

Si nous ajoutons une fonction bloquante comme **confirm()** dans la séquence :

```
console.log("A");
console.log("B");
console.log("C");
console.log("D");
confirm("Voulez-vous continuer l'exécution du code ?");
console.log("E");
console.log("F");
console.log("G");
console.log("H");
```

L'exécution du script s'arrêtera au niveau de la boîte de dialogue **confirm()**, et les instructions suivantes (console.log("E") et suivantes) ne seront exécutées qu'après avoir traité cette fonction. Le navigateur attend donc que la boîte de confirmation soit fermée pour continuer l'exécution du script.



Ce mode de fonctionnement synchrone peut créer des goulots d'étranglement dans l'exécution du programme, surtout si une tâche prend beaucoup de temps (comme un téléchargement de fichier ou une requête réseau), car tout le reste du code sera bloqué en attendant.

JavaScript asynchrone :

En revanche, le mode asynchrone permet à plusieurs tâches de s'exécuter de manière concurrente. Les tâches asynchrones ne bloquent pas l'exécution du reste du code. Cela signifie qu'une tâche longue, comme une requête réseau, peut s'exécuter en arrière-plan

pendant que d'autres tâches continuent de se dérouler. Le JavaScript asynchrone repose sur des mécanismes tels que les callbacks, les promesses, et les fonctions async/await pour gérer ces tâches en parallèle.

L'un des principaux avantages de la programmation asynchrone est qu'elle permet d'améliorer les performances et la réactivité des applications web. Par exemple, une tâche lente, comme le téléchargement d'un fichier ou l'attente d'une réponse d'une API, n'aura pas besoin de bloquer l'interface utilisateur. L'application continuera de répondre aux interactions de l'utilisateur tout en exécutant des opérations en arrière-plan.

Prenons l'exemple d'une application qui doit attendre qu'un fichier soit téléchargé avant de permettre aux autres tâches de s'exécuter. En mode synchrone, cela bloquerait toute l'application, mais en mode asynchrone, d'autres fonctions peuvent s'exécuter en attendant la fin du téléchargement.

Exemple de base d'exécution synchrone

Prenons le fichier index.html suivant pour illustrer ces concepts :

```
    <!DOCTYPE html>
    <html lang="fr">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
            <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <title>JavaScript Asynchrone</title>
    </head>

    <body>
        <script src="my_script.js"></script>
    </body>

    </html>
```

Dans ce fichier, nous avons lié un script externe my_script.js dans lequel nous allons illustrer le comportement synchrone et asynchrone.

Programmation synchrone
Par défaut, toutes les fonctions en JavaScript sont synchrones. Cela signifie que lorsqu'une fonction est appelée :

Elle exécute immédiatement toutes ses instructions et renvoie une valeur.
Le reste du programme attend que cette fonction ait terminé son exécution avant de continuer.
Cela implique qu'en appelant plusieurs fonctions synchrones d'affilée, elles seront exécutées de manière séquentielle, c'est-à-dire l'une après l'autre.

Prenons l'exemple suivant :

```
console.log("A");
console.log("B");
console.log("C");
console.log("D");
console.log("E");
console.log("F");
console.log("G");
console.log("H");
```

Ici, les messages sont affichés successivement dans la console dans l'ordre des instructions, sans interruption.

Un exemple de fonction bloquante
Si nous insérons une fonction bloquante comme confirm() entre deux instructions console.log :

Dans ce cas, l'exécution du script s'arrête à la boîte de dialogue de confirmation. Les instructions suivantes (E, F, etc.) ne seront exécutées qu'après la fermeture de cette boîte de confirmation, ce qui illustre bien le caractère bloquant du mode synchrone.

Transition vers le mode asynchrone
Le mode de fonctionnement synchrone, bien que simple à comprendre, peut devenir une contrainte dans des applications plus complexes, où certaines opérations (comme des requêtes vers un serveur ou des tâches lourdes) peuvent bloquer l'ensemble du programme et dégrader l'expérience utilisateur. Pour surmonter ces limitations, JavaScript propose des outils pour la programmation asynchrone, permettant ainsi une exécution non bloquante et fluide.

Dans la suite de ce chapitre, nous allons voir comment JavaScript permet d'exécuter des fonctions de manière asynchrone, garantissant que les tâches longues ne bloquent pas l'exécution des autres tâches et optimisant ainsi la réactivité de vos applications.

Programmation asynchrone et les fonctions de rappel (callback)

Reprenons notre exemple précédent pour voir comment nous pouvons exécuter toutes les instructions console.log sans bloquer l'exécution avec la boîte de confirmation, grâce à l'utilisation d'une fonction asynchrone.

Exemple 1 :

Nous allons d'abord créer une fonction qui affiche la boîte de confirmation :

```
    function confirmer() {
        confirm("Voulez-vous exécuter la suite du code?");
    }
```

Ensuite, nous utiliserons la fonction JavaScript asynchrone **setTimeout(),** qui prend deux arguments :

- Une fonction de rappel (callback), c'est-à-dire une fonction qui sera exécutée une fois le délai écoulé.
- Un délai en millisecondes, après lequel la fonction de rappel sera exécutée.

Dans notre exemple, la fonction confirmer() sera la fonction de rappel qui sera exécutée après 2 secondes.

Voici le code :

```
    console.log("A");
    console.log("B");
    console.log("C");
    console.log("D");

      // La fonction setTimeout() permet de simuler un délai
d'exécution
```

```
    setTimeout(confirmer, 2000); // Le 1er argument est la fonction à exécuter après expiration du délai.
    console.log("E");
    console.log("F");
    console.log("G");
    console.log("H");

    function confirmer() {
        confirm("Voulez-vous exécuter la suite du code?");
    }
```

Note importante :
Lorsque vous passez une fonction en tant qu'argument (comme c'est le cas avec confirmer dans setTimeout()), il est important de ne pas inclure de parenthèses après le nom de la fonction. En effet, ajouter des parenthèses (confirmer()) exécuterait immédiatement la fonction au lieu de la passer en tant que référence.

Exemple :

```
    setTimeout(confirmer, 2000);
```

Résultat :
Si vous réinitialisez et actualisez la console, vous verrez que toutes les instructions console.log s'affichent immédiatement sans blocage. Puis, après un délai de 2 secondes, la boîte de confirmation apparaît. Cela signifie que la boîte de confirmation n'a pas bloqué l'exécution des instructions suivantes, même si elle a été définie dans le code avant certaines d'entre elles.

Explication :
Lorsque la fonction asynchrone **setTimeout()** est appelée, elle est placée dans une pile de fonctions en attente. Cela permet à JavaScript de continuer à exécuter les autres tâches sans bloquer. Pendant que JavaScript exécute le reste du code, la fonction en attente reste dans cette pile (ou "file d'attente"). Une fois que le délai est expiré (dans ce cas, 2 secondes), la fonction en attente est retirée de la pile et exécutée à son tour. Ce mécanisme fait partie de ce qu'on appelle la "boucle d'événements" ou event loop.

Enchaînement de l'exécution de l'exemple précédent :

- **Exécution des 4 premières instructions console.log** :

Les messages "A", "B", "C", et "D" sont immédiatement affichés dans la console.

- **Appel de la fonction setTimeout** : La fonction asynchrone est appelée, mais placée dans une file d'attente, ce qui permet de continuer l'exécution du reste du code.
- **Exécution des 4 dernières instructions console.log** : Les messages "E", "F", "G", et "H" s'affichent immédiatement, sans attendre la fin du délai.
- **Expiration du délai de 2 secondes** : Après 2 secondes, la fonction setTimeout est retirée de la file d'attente et prête à être exécutée.
- **Exécution de setTimeout** : La fonction de rappel confirmer() est exécutée, et la boîte de confirmation apparaît.

Exemple 2 : Programmation asynchrone avec setTimeout() et un bouton interactif

Étape 1 : Ajout du bouton dans index.html

Nous ajoutons un bouton HTML avec l'identifiant demo dans le fichier index.html :

```
    <button id="demo">Cliquer demo</button>
```

Étape 2 : Script JavaScript dans script.js

Nous allons maintenant ajouter le script qui inclut la fonction asynchrone testerAsynchrne(). Cette fonction est appelée lorsque l'utilisateur clique sur le bouton. Voici le code :

```
function testerAsynchrne() {
    function maFonction() {
      console.log("Premier log");
       setTimeout(() => { console.log('log1 de setTimeout après 0s')
}, 0);
       setTimeout(() => { console.log('log2 de setTimeout après 4s')
}, 4000);
       setTimeout(() => { console.log('log3 de setTimeout après 2s')
}, 2000);
      console.log('Log après les 3 setTimeout!!!');
    }
```

```
      // Ajout d'un événement au clic sur le bouton avec
id="demo"
            document.getElementById('demo').addEventListener('click',
function () {
            console.log('Le programme commence, attachez vos
ceintures');
      maFonction(); // Appel de la fonction
      console.log('Le programme a t-il vraiment terminé?');
    });
}

testerAsynchrne(); // Appel de la fonction principale
```

Résultat dans la console :
Lorsque vous cliquez sur le bouton "Cliquer demo", voici ce qui s'affiche dans la console :

```
Le programme commence, attachez vos ceintures
Premier log
Log après les 3 setTimeout!!!
Le programme a t-il vraiment terminé?
log1 de setTimeout après 0s
log3 de setTimeout après 2s
log2 de setTimeout après 4s
```

Explication détaillée :

Appel de la fonction testerAsynchrne() : Lorsque la page est chargée, la fonction testerAsynchrne() est définie mais n'est pas encore exécutée. Elle sera exécutée lorsqu'on clique sur le bouton avec l'identifiant demo.

Ajout d'un événement de clic : Un écouteur d'événements est ajouté au bouton demo. Lorsque l'utilisateur clique sur ce bouton, la fonction de rappel (callback) est exécutée. Cette fonction contient les instructions suivantes :

- Affiche le message "Le programme commence, attachez vos ceintures".

- Appelle la fonction maFonction().
- Affiche le message "Le programme a t-il vraiment terminé?".

Exécution de maFonction() : Lorsque maFonction() est appelée, elle effectue les actions suivantes :

- Affiche "Premier log" dans la console.
- Appelle trois fois la fonction setTimeout() pour créer des temporisateurs asynchrones :
  - Le premier setTimeout() est exécuté avec un délai de 0 millisecondes, ce qui signifie qu'il sera exécuté dès que possible après l'exécution du reste du code synchrone.
  - Le second setTimeout() est exécuté après un délai de 4000 millisecondes (4 secondes).
  - Le troisième setTimeout() est exécuté après un délai de 2000 millisecondes (2 secondes).

- Après avoir défini les trois temporisateurs setTimeout(), la fonction affiche immédiatement "Log après les 3 setTimeout!!!". Cela montre que JavaScript continue à exécuter le code sans attendre l'expiration des délais.

Exécution de setTimeout() :

- Le premier setTimeout() (0 millisecondes) s'exécute juste après l'exécution du code synchrone. Il affiche "log1 de setTimeout après 0s" dans la console.
- Le second setTimeout() (2 secondes) s'exécute après un délai de 2 secondes et affiche "log3 de setTimeout après 2s".
- Le troisième setTimeout() (4 secondes) s'exécute après 4 secondes et affiche "log2 de setTimeout après 4s".

Résumé :

Cet exemple montre comment la fonction setTimeout() permet d'exécuter du code de manière asynchrone. JavaScript continue d'exécuter les instructions après avoir défini les temporisateurs, et ces temporisateurs sont placés dans une file d'attente (appelée Event Loop). Les tâches asynchrones ne bloquent pas l'exécution du code synchrone, ce qui permet d'optimiser les performances et de rendre les applications plus réactives.

Exemple 3 : Chargement asynchrone d'une image avec

addEventListener

Imaginons que vous souhaitiez afficher une image (ou une vidéo) sur votre page web. Si vous utilisez une approche synchrone, le navigateur devra attendre que l'image soit entièrement chargée avant de poursuivre l'exécution des autres instructions JavaScript. Pendant ce temps, l'utilisateur pourrait subir un blocage ou un ralentissement, affectant l'expérience de navigation.

En revanche, avec une approche asynchrone, le navigateur peut continuer à exécuter d'autres instructions pendant que l'image est chargée en arrière-plan. Cela permet de maintenir une interface fluide et réactive. Pour cela, vous pouvez utiliser la méthode addEventListener() de JavaScript, qui permet de détecter lorsqu'une image est complètement chargée et d'exécuter du code en conséquence.

Voici comment cela fonctionne :

```
    // Création d'un élément <img>
    const img = document.createElement('img');

    // Définition de la source de l'image
    img.src = 'images/image.jpg';

       // Ajout d'un écouteur d'événement pour détecter le
chargement de l'image
    img.addEventListener('load', () => {
        // L'image est maintenant complètement chargée
        // Nous pouvons l'ajouter au DOM pour l'afficher
        document.body.appendChild(img);
    });
```

Explication :

- **Création de l'élément image :**

La première ligne de ce code crée **dynamiquement** un nouvel élément HTML <img> sans l'ajouter immédiatement au DOM. Cela permet de définir les propriétés de l'image avant qu'elle ne soit affichée.

- **Définition de la source de l'image :**

La deuxième ligne de ce code permet de définir la source de l'image (son url). Le navigateur commence automatiquement à charger l'image en arrière-plan, mais nous ne bloquons pas l'exécution des autres parties du script.

- **Écoute de l'événement load :**

Le dernier bloc de ce code permet de déclencher l'événement **load** une fois que l'image a terminé son chargement. Grâce à **addEventListener()**, nous définissons un écouteur d'événement qui exécute une fonction lorsque l'image est **complètement** chargée. La fonction fléchée passée en argument est appelée de manière asynchrone. Dès que l'image est prête, elle est ajoutée au document HTML avec **appendChild()**, permettant son affichage.

Ainsi, le chargement de l'image se fait en arrière-plan et le reste du code JavaScript continue à s'exécuter sans interruption.

Ajout d'un autre élément synchrone :
Pendant que l'image se charge, d'autres parties du code peuvent continuer à s'exécuter. Par exemple, nous pouvons modifier le contenu d'un paragraphe sans attendre la fin du chargement de l'image. Supposons que vous avez dans votre page index.html, un paragraphe ayant comme id = parag et vous souhaitez modifier son contenu en parallèle du chargement de l'image.
Code à ajouter dans index.html :

```
<p id="parag">Je suis un simple paragraphe et j'attends vos modifications</p>
```

Ajoutons par exemple le code suivant à la fin du script myscript.js

```
const myParag = document.getElementById("parag");
myParag.textContent = "Bienvenue sur notre site web. Nous vous remercions de votre visite ! N'hésitez pas à explorer les différentes sections et à découvrir nos services.";
```

Ici, nous modifions immédiatement le contenu du paragraphe avec l'ID parag, tandis que l'image continue de se charger en arrière-plan.

En résumé, dans cet exemple, l'utilisation de l'événement **load** nous permet de charger une image de manière asynchrone. L'image est ajoutée à la page web une fois qu'elle est complètement chargée, sans

bloquer l'exécution du reste du script. Cette approche permet d'améliorer les performances et la fluidité des applications web, en garantissant que les utilisateurs ne soient pas confrontés à des blocages ou des ralentissements lors du chargement des ressources.

Exemple 4 : Utilisation de la fonction setInterval en JavaScript

La fonction setInterval en JavaScript est utilisée pour exécuter une fonction de manière répétée, à des intervalles réguliers définis en millisecondes. En d'autres termes, cette fonction permet d'appeler une autre fonction périodiquement, à chaque intervalle de temps spécifié.

Syntaxe générale de setInterval :

```
    setInterval(function, time);
```

- **function** : La fonction que vous souhaitez exécuter périodiquement.
- **time** : L'intervalle en millisecondes entre chaque exécution de la fonction.

Lorsque vous appelez **setInterval(),** la fonction spécifiée sera exécutée en boucle, à l'intervalle de temps défini, jusqu'à ce que vous arrêtiez cette répétition en appelant la fonction **clearInterval().**

Exemple pratique : Affichage de l'heure en temps réel
Imaginons que vous souhaitiez afficher l'heure actuelle sur une page web, mise à jour toutes les secondes. Voici comment vous pouvez utiliser setInterval() pour y parvenir.

Étape 1 : Ajout de l'élément HTML dans index.html
Dans le fichier index.html, ajoutez un élément <h1> avec un id qui contiendra l'heure :

```
   <h1 id="dateofthisday"></h1>
```

Étape 2 : Script JavaScript dans script.js
Dans le fichier my_script.js, ajoutez le script suivant pour afficher et mettre à jour l'heure toutes les secondes :

```
     // Sélection de l'élément <h1> dans lequel l'heure sera
affichée
                                let        dateofthisday        =
document.getElementById("dateofthisday");
```

```
    // Fonction pour afficher la date et l'heure actuelles
    function displayDate() {
          let d = new Date(); // Création d'un objet Date pour
obtenir l'heure actuelle
        dateofthisday.innerHTML =
            d.getHours() + ":" +
            d.getMinutes() + ":" +
               d.getSeconds(); // Affichage de l'heure sous la
forme HH:MM:SS
    }

        // Appel de la fonction setInterval qui exécute
afficherDate toutes les 1000ms (1 seconde)
    setInterval(displayDate, 1000);
```

Explication :

Dans la première ligne de ce code, nous récupérons l'élément <h1> grâce à son identifiant `dateofthisday`. Cet élément contiendra l'heure qui sera mise à jour toutes les secondes.

Définition de la fonction displayDate() :

Dans ce code, nous avons défini la fonction displayDate() qui crée un objet **Date** pour obtenir l'heure actuelle et met à jour le contenu de l'élément <h1> avec les heures, minutes et secondes. La méthode **innerHTML** est utilisée pour insérer ce texte dans l'élément.

Appel de setInterval() :

La dernière ligne de ce code appelle la fonction **setInterval()** pour appeler à son tour la fonction **dispalyDate()** toutes les **1000 millisecondes** (1 seconde). Cela permet de rafraîchir l'affichage de l'heure chaque seconde.

Résultat :
L'heure actuelle sera affichée dans l'élément <h1> et sera mise à jour chaque seconde. Vous verrez ainsi les secondes changer en temps réel, grâce à l'appel périodique de la fonction displayDate().

Résumé :
Cet exemple illustre comment JavaScript permet de gérer des tâches répétitives à l'aide de la fonction setInterval(). La programmation asynchrone permet d'exécuter des tâches en arrière-plan ou à des intervalles réguliers sans bloquer l'exécution du reste du code. Cependant, bien que puissante, la gestion des tâches asynchrones peut devenir complexe, surtout lorsqu'on commence à imbriquer plusieurs fonctions de rappel (callbacks) ou à gérer plusieurs tâches asynchrones en parallèle.

Dans les prochaines sections, nous approfondirons ces concepts et verrons comment améliorer la gestion du code asynchrone, en particulier avec l'utilisation des **promesses** et de **async/await** pour rendre le code plus lisible et plus facile à déboguer.

Les promesses en JavaScript

Une promesse (ou Promise en anglais) est un objet en JavaScript qui représente la valeur d'une opération asynchrone. Cette valeur peut être disponible maintenant, plus tard (dans le futur), ou jamais. Les promesses ont été introduites avec la norme ES6 (ECMAScript 2015) pour simplifier l'écriture et la gestion des fonctions asynchrones, remplaçant les nombreux callbacks imbriqués qui compliquaient souvent le code.

Les états d'une promesse :
Une promesse peut être dans l'un des trois états suivants :

- En attente (pending) : L'opération asynchrone est en cours d'exécution, et la promesse n'a pas encore donné de résultat.
- Résolue (fulfilled) : L'opération asynchrone s'est terminée avec succès, et la promesse est résolue avec une valeur (généralement le résultat de l'opération).
- Rejetée (rejected) : L'opération asynchrone a échoué, et la promesse est rejetée avec une raison spécifique (généralement un message d'erreur).

Création d'une promesse :
Pour créer une promesse, vous devez utiliser le constructeur Promise, qui prend en paramètre une fonction. Cette fonction reçoit deux arguments : resolve (appelé lorsque l'opération est réussie) et reject (appelé en cas d'échec).

Exemple pratique :
Voici un exemple qui simule une opération asynchrone en générant un nombre aléatoire. Si ce nombre est supérieur ou égal à 0.5, la promesse est résolue avec ce nombre. Sinon, elle est rejetée avec un message d'erreur. Saisissons le code suivant dans my_script.js :

```
    const myPromise = new Promise(function (resolve, reject) {
        // Simulation d'une opération asynchrone
        setTimeout(function () {
              const numberAleatoire = Math.random(); // Génère
un nombre aléatoire entre 0 et 1
            if (numberAleatoire >= 0.5) {
                 resolve(numberAleatoire); // Promesse résolue
avec le nombre aléatoire
            } else {
                reject("Le nombre trouvé " + numberAleatoire +
" est inférieur à 0.5");
            }
        }, 1000); // Délai simulé de 1 seconde
    });
```

Explication :

Dans cet exemple :

- Math.random() génère un nombre aléatoire entre 0 et 1.
- Si le nombre généré est supérieur ou égal à 0.5, la promesse est résolue.
- Si le nombre est inférieur à 0.5, la promesse est rejetée avec un message d'erreur spécifique.

Utilisation des méthodes then() et catch() :
Une fois la promesse créée, nous utilisons les méthodes **then()** et **catch()** pour gérer respectivement le succès et l'échec de la promesse. Ajoutons le code suivant dans my_script.js :

```
myPromise.then(function (result) {
     console.log("La promesse a été résolue avec succès : " +
result);
}).catch(function (raison) {
```

```
    console.log("La promesse a été rejetée avec la raison
suivante : " + raison);
});
```

- then() : Cette méthode est appelée lorsque la promesse est résolue avec succès. Elle reçoit le résultat de la promesse (dans notre cas, le nombre aléatoire).
- catch() : Cette méthode est appelée lorsque la promesse est rejetée. Elle reçoit la raison du rejet (ici, un message d'erreur indiquant que le nombre est inférieur à 0.5).

Résultat attendu :
Si vous exécutez ce code, vous obtiendrez l'un des deux résultats suivants, après un délai de 1 seconde (1000 millisecondes) :

Si le nombre aléatoire est supérieur ou égal à 0.5 :

```
La promesse a été résolue avec succès : 0.8801144648302703
```

Si le nombre aléatoire est inférieur ou égal à 0.5 :

```
La promesse a été rejetée avec la raison suivante : Le nombre trouvé
0.353270848072264 83 est inférieur à 0.5
```

En conclusion, les promesses sont un outil puissant pour gérer les opérations asynchrones en JavaScript de manière plus fluide et plus lisible que les simples callbacks imbriqués. En encapsulant le code asynchrone dans des objets Promise, nous pouvons facilement gérer le succès et l'échec des opérations à l'aide des méthodes then() et catch(). Cela rend le code plus propre, plus modulaire, et plus facile à comprendre, tout en facilitant la gestion des erreurs.

Dans les sections suivantes, nous explorerons des fonctionnalités encore plus puissantes de JavaScript moderne, telles que **async/ await**, qui simplifient encore plus la gestion des promesses et rendent le code asynchrone plus proche du code synchrone classique.

Async et Await en JavaScript

Les mots-clés async et await ont été introduits dans ES2017 (ou ES8) pour simplifier la gestion des promesses et rendre le code asynchrone plus lisible et plus facile à écrire. Ils permettent de travailler avec des fonctions asynchrones de manière plus proche du style synchrone

traditionnel.

async :
Le mot-clé async est utilisé pour déclarer une fonction asynchrone. Lorsqu'une fonction est définie avec async, elle retourne automatiquement une promesse. Cela signifie que la fonction suspend son exécution jusqu'à ce que la promesse soit résolue ou rejetée. Cela permet de gérer les opérations asynchrones de manière beaucoup plus claire.

await :
Le mot-clé await ne peut être utilisé qu'à l'intérieur d'une fonction async. Il est utilisé pour attendre la résolution d'une promesse avant de continuer l'exécution du reste du code dans la fonction. Il simplifie la gestion des promesses en permettant d'éviter d'utiliser explicitement les méthodes then() et catch().

Voyons comment utiliser async et await pour réécrire l'exemple précédent sur les promesses.

Exemple avec async et await :
Voici comment nous pouvons réécrire l'exemple des promesses en utilisant les fonctionnalités async et await :

```
    async function myFunction() {
        try {
                const numberRandom = await new Promise(function
(resolve, reject) {
                // Simulation d'une opération asynchrone
                setTimeout(function () {
                    const number = Math.random();
                    if (number >= 0.5) {
                        resolve(number); // Promesse résolue
                    } else {
                        reject("Le nombre trouvé " + number +
" est inférieur à 0.5"); // Promesse rejetée
                    }
                }, 1000); // Délai simulé de 1 seconde
            });

            // Si la promesse est résolue avec succès
            console.log("La promesse a été résolue avec succès
```

```
: " + numberRandom);
        } catch (error) {
            // Si la promesse est rejetée
                console.log("La promesse a été rejetée avec la
raison suivante : " + error);
        }
    }

    // Appel de la fonction
    myFunction();
```

Explication :

Déclaration de la fonction avec async :

```
  async function myFunction() {
        // Code asynchrone ici
    }
```

La fonction **myFunction()** est déclarée comme une fonction asynchrone grâce au mot-clé **async**. Cela signifie que cette fonction retourne une promesse et que son exécution sera suspendue jusqu'à ce que les promesses internes soient résolues ou rejetées.

Utilisation de await pour attendre la résolution de la promesse :

```
const numberRandom = await new Promise(function (resolve,
reject) {
        // Simulation d'une opération asynchrone
    });
```

À l'intérieur de la fonction **myFunction()**, nous utilisons **await** pour attendre que la promesse soit résolue. Cela signifie que le code ne continuera pas à s'exécuter tant que la promesse ne renvoie pas une valeur. Ici, nous simulons une opération asynchrone avec **setTimeout()** qui attend 1 seconde avant de générer un nombre aléatoire.

Simulation d'une opération asynchrone :

```
   setTimeout(function () {
        const number = Math.random();
```

```
        if (number >= 0.5) {
            resolve(number); // Promesse résolue
        } else {
                reject("Le nombre trouvé " + number + " est
inférieur à 0.5"); // Promesse rejetée
        }
    }, 1000);
```

L'opération asynchrone est simulée par setTimeout(), qui attend 1 seconde avant de générer un nombre aléatoire avec Math.random(). Si ce nombre est supérieur ou égal à 0.5, la promesse est résolue avec ce nombre. Sinon, elle est rejetée avec un message d'erreur.

Gestion des résultats avec try...catch :

- Si la promesse est résolue avec succès, la valeur de numberRandom est affichée dans la console :

```
      console.log("La promesse a été résolue avec succès : " +
numberRandom);
```

- Si la promesse est rejetée, l'erreur est capturée et affichée dans la console avec un message explicatif :

```
console.log("La promesse a été rejetée avec la raison suivante
: " + error);
```

Résultat attendu :
Comme dans l'exemple précédent sur les promesses, l'exécution de ce code aboutira à l'un des deux résultats suivants, après une seconde d'attente :

- Si le nombre généré est supérieur ou égal à 0.5 :

```
La promesse a été résolue avec succès : 0.9335581072777603
```

- Si le nombre généré est inférieur à 0.5 :

```
La promesse a été rejetée avec la raison suivante : Le nombre trouvé
0.22352467896205286 est inférieur à 0.5
```

En conclusion, l'utilisation des mots-clés **async** et **await** simplifie considérablement la gestion des opérations asynchrones en JavaScript. Ils permettent de rendre le code asynchrone plus lisible et plus proche

du code synchrone classique. Avec **async**, les fonctions asynchrones retournent automatiquement des promesses, et await permet d'attendre leur résolution sans avoir à recourir aux méthodes **then()** et **catch()**. Cela permet également de gérer les erreurs de manière plus fluide avec **try...catch**, facilitant ainsi le débogage et la maintenance du code.

Cet exemple montre que **async/await** aboutit au même résultat que l'utilisation classique des promesses, mais avec un code beaucoup plus propre et plus intuitif.

Les promesses et le Callback Hell

Callback Hell

Le "callback hell" est une situation où plusieurs fonctions asynchrones sont imbriquées les unes dans les autres, créant un code difficile à lire, à comprendre et à maintenir. Cette structure enchaînée de callbacks produit une indentation excessive et une organisation de code complexe, souvent comparée à une "pyramide de la mort".

Exemple de Callback Hell :
Dans cet exemple, nous avons 7 fonctions asynchrones qui se suivent les unes après les autres à l'aide de callbacks. Chaque fonction attend que la précédente se termine avant de pouvoir s'exécuter.

```
function firstFunction(callback) {
    setTimeout(() => {
        console.log("First function");
        callback();
    }, 1000);
}

function secondFunction(callback) {
    setTimeout(() => {
        console.log("Second function");
        callback();
    }, 1000);
}

function thirdFunction(callback) {
    setTimeout(() => {
```

```
            console.log("Third function");
            callback();
        }, 1000);
    }

    function fourthFunction(callback) {
        setTimeout(() => {
            console.log("Fourth function");
            callback();
        }, 1000);
    }

    function fifthFunction(callback) {
        setTimeout(() => {
            console.log("Fifth function");
            callback();
        }, 1000);
    }

    function sixthFunction(callback) {
        setTimeout(() => {
            console.log("Sixth function");
            callback();
        }, 1000);
    }

    function seventhFunction(callback) {
        setTimeout(() => {
            console.log("Seventh function");
            callback();
        }, 1000);
    }

    // Appel des fonctions de manière imbriquée
firstFunction(function () {
    secondFunction(function () {
        thirdFunction(function () {
            fourthFunction(function () {
                fifthFunction(function () {
```

```
                        sixthFunction(function () {
                            seventhFunction(function () {
                                    console.log('End of the "callback
hell"!');
                                });
                            });
                        });
                    });
                });
            });
        });
```

Explication:

- Chaque fonction, de **firstFunction** à **seventhFunction**, utilise la méthode **setTimeout()** pour simuler une opération asynchrone (ici un délai d'une seconde avant l'exécution de la fonction).
- À l'intérieur de chaque fonction, le **callback** est appelé après l'exécution de console.log().
- Les fonctions sont ensuite imbriquées les unes dans les autres, chacune attendant que la précédente termine avant de s'exécuter. Cela conduit à une structure pyramidale de **callbacks imbriqués**, créant ce qu'on appelle le "**callback hell**".

Problème du Callback Hell :
Le principal problème de ce code est qu'il est difficile à lire et à maintenir. Avec chaque nouvelle fonction ajoutée, l'indentation devient plus profonde, rendant le code moins lisible et plus complexe à déboguer. De plus, si chaque callback contenait plus de logique, cela amplifierait le problème et la difficulté de compréhension.

Solutions pour éviter le Callback Hell :
Heureusement, il existe des solutions pour éviter ces structures pyramidales et rendre le code plus lisible et maintenable :

- **Les promesses** : En utilisant des promesses, vous pouvez "chaîner" les appels de fonctions asynchrones, ce qui réduit l'indentation et rend le code plus linéaire.
- **async/await** : Avec **async/await**, vous pouvez écrire du code asynchrone qui ressemble à du code synchrone,

améliorant encore la lisibilité.

Les promesses et la résolution des contraintes du Callback Hell

Avec les promesses, il est possible de chaîner les opérations asynchrones de manière plus lisible. Cela permet de les écrire les unes après les autres, tout en les exécutant de manière séquentielle. L'idée clé est que chaque fonction s'exécute après la résolution de la promesse précédente. Cela résout le problème de lisibilité rencontré avec le callback hell.

Exemple avec des promesses :

```
function firstFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("First function");
            resolve();
        }, 1000);
    });
}

function secondFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Second function");
            resolve();
        }, 1000);
    });
}

function thirdFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Third function");
            resolve();
        }, 1000);
    });
}
```

```
function fourthFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Fourth function");
            resolve();
        }, 1000);
    });
}

function fifthFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Fifth function");
            resolve();
        }, 1000);
    });
}

function sixthFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Sixth function");
            resolve();
        }, 1000);
    });
}

function seventhFunction() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Seventh function");
            resolve();
        }, 1000);
    });
}

// Chaînage des promesses
firstFunction()
```

```
    .then(secondFunction)
    .then(thirdFunction)
    .then(fourthFunction)
    .then(fifthFunction)
    .then(sixthFunction)
    .then(seventhFunction)
    .then(function () {
        console.log("End of promises!");
    });
```

Explication :

- **Fonctions avec des promesses** : Chaque fonction, de **firstFunction** à **seventhFunction**, renvoie une promesse. Cette promesse est résolue à la fin d'une opération asynchrone simulée avec **setTimeout().** Par exemple, dans **firstFunction(),** la promesse est résolue après 1 seconde d'attente, et la fonction suivante peut alors être exécutée.

- **Chaînage des promesses avec then()** : En utilisant la méthode **then(),** nous chaînons les appels des fonctions les unes après les autres. Cela garantit que chaque fonction asynchrone s'exécute séquentiellement après la résolution de la promesse précédente. Par exemple, **secondFunction()** est appelée après que **firstFunction()** ait été résolue, et ainsi de suite.

- **Affichage final** : À la fin du chaînage, nous ajoutons un dernier **then()** qui affiche un message indiquant que toutes les promesses ont été résolues et que toutes les fonctions asynchrones ont été exécutées avec succès.

Avantages de l'utilisation des promesses :

- **Meilleure lisibilité** : Comparé à l'imbrication excessive des callbacks (callback hell), le chaînage des promesses permet d'écrire du code plus lisible et mieux structuré.
- **Exécution séquentielle** : Les promesses permettent de garantir que chaque fonction est exécutée après la résolution de la promesse précédente, en respectant l'ordre d'exécution.
- **Gestion des erreurs facilitée** : En ajoutant un bloc catch() à la fin du chaînage des promesses, il est

également possible de gérer facilement les erreurs, ce qui n'était pas toujours évident avec les callbacks.

En conclusion, dans cet exemple, nous avons utilisé des promesses pour chaque fonction asynchrone (comme firstFunction(), secondFunction(), etc.). Chaque fonction renvoie une promesse qui est résolue après l'exécution d'une opération asynchrone simulée. En chaînant ces promesses avec la méthode **then(),** nous avons évité l'imbrication complexe des callbacks (**callback hell**), rendant ainsi le code plus lisible et plus facile à maintenir.

Enfin, après la résolution de toutes les promesses, nous affichons un message pour indiquer que toutes les opérations asynchrones sont terminées.

Conclusion

La programmation asynchrone est un pilier fondamental de JavaScript moderne. Grâce aux mécanismes tels que les callbacks, les promesses, et les fonctions async/await, JavaScript offre une grande flexibilité pour gérer des opérations complexes comme les appels réseaux, le traitement de fichiers, ou l'interaction avec des API, tout en maintenant la fluidité de l'interface utilisateur.

Au fil de ce chapitre, nous avons exploré différentes manières de gérer l'asynchronie :

Nous avons commencé par comprendre les callbacks, la méthode originelle d'exécution des tâches asynchrones en JavaScript, tout en découvrant les pièges du fameux callback hell.
Nous avons ensuite introduit les promesses, qui permettent de chaîner des opérations asynchrones de manière plus élégante, tout en simplifiant la gestion des erreurs.
Enfin, nous avons abordé les async/await, un des outils les plus puissants et lisibles pour écrire du code asynchrone, qui rend le code plus linéaire et plus proche de l'approche synchrone tout en conservant les avantages de l'asynchronie.
Ces outils, bien que puissants, nécessitent une bonne compréhension pour être utilisés efficacement. En maîtrisant la programmation asynchrone, vous pouvez développer des applications web plus rapides, plus réactives et capables de gérer efficacement les opérations longues ou bloquantes, sans dégrader l'expérience utilisateur.

L'asynchronie en JavaScript est une compétence essentielle pour tout développeur moderne. Ce chapitre vous a permis de comprendre les

principes fondamentaux et les différentes techniques pour gérer les tâches asynchrones. Il ne vous reste plus qu'à pratiquer ces concepts pour les maîtriser pleinement.

Chapitre 19

Requêtes HTTP avec AJAX et Fetch API

Dans le développement web moderne, la capacité à communiquer avec un serveur est cruciale pour créer des applications interactives et dynamiques. Que ce soit pour récupérer des données depuis une API, soumettre des formulaires, ou encore mettre à jour une page sans la recharger entièrement, JavaScript nous offre des outils puissants pour effectuer des requêtes HTTP asynchrones.

Historiquement, les développeurs ont utilisé AJAX (Asynchronous JavaScript and XML) pour accomplir ces tâches. AJAX a révolutionné la manière dont les applications web interagissent avec les serveurs en permettant de charger du contenu dynamiquement, sans nécessiter de rechargement complet de la page. C'est grâce à AJAX que des applications modernes comme Google Maps ou Gmail ont pu offrir une expérience utilisateur fluide et réactive.

Cependant, avec l'évolution des technologies web, une nouvelle API a émergé pour simplifier davantage la gestion des requêtes HTTP : la Fetch API. Contrairement à AJAX, qui repose sur l'objet XMLHttpRequest, la Fetch API propose une syntaxe plus moderne et plus facile à utiliser, tout en offrant plus de flexibilité pour travailler avec les promesses et l'asynchronie en JavaScript.

Dans ce chapitre, nous allons explorer :

- Les bases des requêtes HTTP.
- Comment utiliser AJAX pour envoyer et recevoir des données de manière asynchrone.
- La transition vers la Fetch API et pourquoi elle est désormais le standard moderne pour effectuer des requêtes réseau.
- Des exemples pratiques pour illustrer comment intégrer ces concepts dans vos applications web.

Maîtriser les requêtes HTTP est essentiel pour tout développeur souhaitant créer des applications riches et interactives. Préparez-vous à découvrir comment ces outils vous permettront de connecter vos applications aux serveurs, de manipuler les données et d'offrir une expérience utilisateur moderne et efficace.

Les Requêtes HTTP : Un Aperçu

Avant de plonger dans les détails d'AJAX et de la Fetch API, il est important de comprendre les bases des requêtes HTTP. Le protocole HTTP (Hypertext Transfer Protocol) est le fondement des communications entre les navigateurs et les serveurs web. Il permet d'envoyer des demandes (requêtes) et de recevoir des réponses.

Les requêtes HTTP sont généralement utilisées pour :

- Récupérer des données depuis un serveur (via des APIs, fichiers JSON, etc.).
- Envoyer des données au serveur (comme des soumissions de formulaires).
- Mettre à jour une partie de la page web sans recharger toute la page (par exemple, rafraîchir les résultats de recherche en temps réel).

Structure d'une requête HTTP
Une requête HTTP se compose de plusieurs parties :

- **Méthode HTTP** : Elle définit l'action que vous souhaitez effectuer. Les plus courantes sont :
  - GET : Pour récupérer des données depuis un serveur.
  - POST : Pour envoyer des données au serveur.
  - PUT : Pour mettre à jour une ressource existante sur le serveur.
  - DELETE : Pour supprimer une ressource du serveur.

- **URL** : L'adresse (ou endpoint) où la requête est envoyée. Par exemple, une API pourrait avoir une URL telle que : https://api.exemple.com/utilisateurs.

- **Headers (En-têtes)** : Des informations supplémentaires qui accompagnent la requête, comme le type de contenu que vous envoyez ou acceptez (ex : Content-Type: application/json).

- **Corps de la requête (Request Body)** : Utilisé principalement avec les requêtes POST pour envoyer des données au serveur (ex : des informations de formulaire ou des objets JSON).

Qu'est-ce que JSON ?

JSON (JavaScript Object Notation) est un format de données léger utilisé pour structurer et échanger des informations. Il est conçu pour être à la fois facile à lire et à écrire pour les humains, et facile à analyser et à générer pour les machines. JSON est principalement utilisé pour échanger des données entre un serveur et un client, en particulier dans les applications web.

Contrairement à des formats plus complexes comme XML, JSON est une alternative plus simple et efficace. Alors qu'XML était autrefois le format standard pour l'échange de données, JSON est devenu la norme aujourd'hui en raison de sa légèreté et de sa compatibilité avec presque tous les langages de programmation modernes.

Les fichiers JSON sont généralement enregistrés avec l'extension .json, et leur syntaxe repose sur des objets et des tableaux similaires à ceux utilisés dans JavaScript, bien que JSON soit devenu un format indépendant du langage.

La syntaxe JSON
JSON repose sur une structure simple avec :

- 4 types primitifs : les nombres, les chaînes de caractères (textes), les booléens (true ou false), et la valeur null.
- 2 types complexes : les objets (collections de paires clé/valeur) et les tableaux (listes ordonnées de valeurs).

Bien que le JSON soit issu de JavaScript, il est compatible avec de nombreux autres langages de programmation tels que Python, C, PHP, et bien d'autres, ce qui en fait un format d'échange de données universel.

Exemple de code JSON :

```
{
    "name": "Jean",
    "age": 30,
    "city": "Paris",
    "country": "France"
}
```

Dans cet exemple :

- "name", "age", "city", et "country" sont les clés.
- "Jean", 30, "Paris", et "France" sont les valeurs correspondantes.

Les clés dans JSON doivent toujours être des chaînes de caractères (enfermées entre guillemets doubles), tandis que les valeurs peuvent être des chaînes de caractères, des nombres, des booléens, des tableaux, ou encore des objets JSON.

Grâce à sa simplicité et à sa lisibilité, JSON est aujourd'hui indispensable pour le développement d'applications modernes, notamment dans le cadre des communications avec des APIs.

AJAX : Historique et Utilisation

Le terme **AJAX** (Asynchronous JavaScript and XML) fait référence à un ensemble de techniques qui permettent d'envoyer et de recevoir des données en arrière-plan sans recharger la page entière. Bien que son nom inclue "XML", il est couramment utilisé pour manipuler des formats de données plus récents comme JSON.

Comment fonctionne AJAX ?
AJAX repose principalement sur l'objet XMLHttpRequest. Cet objet permet de créer des requêtes HTTP en JavaScript et de les envoyer au serveur. Voici un exemple d'utilisation d'AJAX pour effectuer une requête GET afin de récupérer des données d'une API.

Exemple d'utilisation de l'objet XMLHttpRequest

```
    const xhr = new XMLHttpRequest();
       xhr.open("GET",  "https://api.exemple.com/utilisateurs",
true);  // true pour rendre la requête asynchrone
    xhr.onload = function () {
        if (xhr.status >= 200 && xhr.status < 300) {
            const utilisateurs = JSON.parse(xhr.responseText);
             console.log(utilisateurs);  // Affiche les données
reçues
        } else {
                    console.error("Erreur lors de la requête",
xhr.status);
        }
    };
    xhr.onerror = function () {
        console.error("Erreur réseau");
    };
    xhr.send();  // Envoi de la requête
```

Explication:

- Création d'un objet XMLHttpRequest : Nous commençons par créer une instance de l'objet XMLHttpRequest.
- Ouverture de la requête : La méthode open() prépare la requête. Nous spécifions la méthode HTTP (ici GET), l'URL de l'API, et true pour indiquer que la requête doit être asynchrone.
- Traitement de la réponse : Nous utilisons l'événement onload pour traiter la réponse du serveur lorsque la requête est terminée avec succès (statut HTTP 200 à 299).
- Gestion des erreurs : En cas d'erreur réseau, nous utilisons onerror pour afficher un message dans la console.
- Envoi de la requête : Enfin, la méthode send() envoie effectivement la requête au serveur.

Cet exemple montre comment AJAX nous permet de récupérer des données sans recharger toute la page. Cependant, malgré sa puissance, XMLHttpRequest peut paraître verbeux et parfois peu intuitif, notamment lorsqu'il s'agit de gérer les erreurs ou de traiter des réponses complexes.

Fetch API : La solution moderne

Avec l'introduction de la Fetch API, la gestion des requêtes HTTP en JavaScript est devenue plus simple et plus élégante. La Fetch API offre une syntaxe basée sur les promesses, ce qui la rend plus intuitive et adaptée aux pratiques modernes de programmation asynchrone.

L'une des plus grandes forces de la Fetch API est sa compatibilité native avec les promesses, ce qui simplifie grandement la gestion des requêtes asynchrones.

Exemple de requête GET avec Fetch API
Voici comment réaliser une requête GET avec la Fetch API pour récupérer la même liste d'utilisateurs :

```
fetch("https://api.exemple.com/utilisateurs")
    .then(response => {
```

```
            if (!response.ok) {
                        throw new Error("Erreur HTTP : " +
response.status);
            }
               return response.json(); //Convertit la réponse en
format JSON
        })
        .then(data => {
                  console.log(data);   // Affiche les données
récupérées
        })
        .catch(error => {
              console.error("Erreur lors de la requête Fetch :
", error);
        });
```

Explication:

- **Appel à fetch()** : La méthode fetch() est utilisée pour envoyer la requête. Elle prend l'URL de l'API comme paramètre et renvoie une promesse.
- **Vérification de la réponse** : Dans le premier **.then(),** nous vérifions que le statut de la réponse est correct (code HTTP 200 à 299). Si ce n'est pas le cas, nous levons une erreur.
- **Traitement des données** : Si la réponse est correcte, nous convertissons le corps de la réponse en JSON et le passons au bloc suivant.
- **Gestion des erreurs** : Le **.catch()** est utilisé pour gérer les erreurs de la requête ou du traitement des données.

Pourquoi utiliser Fetch API ?

- **Syntaxe moderne** : Comparée à **XMLHttpRequest**, la syntaxe de **fetch()** est beaucoup plus concise et facile à lire.
- **Utilisation des promesses** : **Fetch API** s'intègre parfaitement avec le modèle de promesses de JavaScript, ce qui permet d'écrire du code asynchrone de manière fluide et plus intuitive.
- **Gestion simplifiée des erreurs** : La gestion des erreurs est plus directe avec la **Fetch API**, grâce à l'utilisation de

promesses et de **blocs .catch().**

L'envoi de données avec une requête POST

Dans les exemples précédents, nous avons utilisé des requêtes GET pour récupérer des informations à partir d'un serveur. Cependant, il est tout aussi courant, voire plus important dans certaines applications, d'envoyer des données à un serveur. Pour ce faire, nous utilisons généralement des requêtes POST.

Une requête POST permet d'envoyer des données, comme des informations d'un formulaire ou un objet JSON, au serveur. Contrairement à une requête GET, où les paramètres sont inclus dans l'URL, les données envoyées avec une requête POST sont généralement placées dans le corps de la requête.

Exemple d'envoi de données avec fetch() et une requête POST
Voici un exemple d'envoi d'un objet JSON au serveur avec une requête POST en utilisant la Fetch API.

```
    // Création des données à envoyer
    const user = {
        name: "Jean",
        age: 30,
        city: "Paris",
        country: "France"
    };

    // Envoi de la requête POST
    fetch("https://api.exemple.com/users", {
        method: "POST",  // Spécification de la méthode POST
        headers: {
              "Content-Type": "application/json"  // Indication
que nous envoyons du JSON
        },
          body: JSON.stringify(utilisateur)   // Conversion de
l'objet JavaScript en chaîne JSON
    })
        .then(response => {
            if (!response.ok) {
```

```
                    throw new Error("Erreur HTTP : " +
response.status);
                }
                    return response.json();   // Conversion de la
réponse en JSON
            })
            .then(data => {
                console.log("Données reçues du serveur :", data);
// Traitement de la réponse du serveur
            })
            .catch(error => {
                    console.error("Erreur lors de l'envoi de la
requête POST :", error);
            });
```

Explication :

- **Les données à envoyer** : Nous créons un objet JavaScript appelé « user » contenant les informations que nous voulons envoyer au serveur.
- **La méthode POST** : Dans la méthode **fetch(),** nous spécifions la méthode **POST** avec l'option **method: "POST".** Cela indique que nous envoyons des données au serveur.
- **Les en-têtes de la requête** : Nous définissons les en-têtes avec headers, en particulier **Content-Type: application/json**, pour indiquer au serveur que nous envoyons des données sous forme de **JSON**.
- **Le corps de la requête** : Nous utilisons la méthode **JSON.stringify()** pour convertir l'objet user en chaîne **JSON**, car le corps d'une requête **HTTP** doit être en texte.
- **Traitement de la réponse** : Une fois la requête envoyée, nous traitons la réponse en vérifiant d'abord si la requête s'est bien déroulée (vérification du statut **HTTP**), puis en convertissant la réponse en **JSON**. Si tout se passe bien, nous affichons les données reçues dans la console. En cas d'échec, nous gérons l'erreur avec **.catch().**

Requête POST vs GET

- **GET** est utilisé pour récupérer des données depuis le serveur. Les données envoyées dans une requête GET (s'il y en a) sont généralement placées dans l'URL sous forme de paramètres de requête.
- **POST** est utilisé pour envoyer des données au serveur, souvent pour créer ou mettre à jour des ressources. Les données sont placées dans le corps de la requête, ce qui permet d'envoyer de grandes quantités d'informations, souvent sous forme de JSON.

Meilleures pratiques pour utiliser Fetch API
Maintenant que nous avons vu comment effectuer des requêtes **GET** et **POST** avec **Fetch API**, voici quelques meilleures pratiques à suivre lors de son utilisation dans vos projets.

- **Gérer les erreurs de manière efficace**

Lorsqu'on travaille avec des requêtes réseau, il est important de toujours gérer les erreurs. Les erreurs peuvent provenir de différents endroits :

- **Erreurs réseau** (le serveur n'est pas joignable).
- **Erreurs HTTP** (comme 404 pour une page non trouvée, ou 500 pour une erreur serveur).
- Pr**oblèmes de traitement** (conversion en JSON qui échoue, etc.).

La Fetch API ne rejette pas les promesses en cas de réponse avec un statut d'erreur HTTP (comme 404 ou 500). Vous devez donc vérifier manuellement si la réponse est correcte en utilisant response.ok.

```
    fetch("https://api.exemple.com/users")
        .then(response => {
            if (!response.ok) {
                        throw new Error("Erreur HTTP : " +
response.status);
            }
            return response.json();
        })
        .catch(error => {
                console.error("Erreur lors de la requête :",
error);
```

```
        });
```

- **Utiliser async/await pour une meilleure lisibilité**

La syntaxe **async/await** rend les requêtes asynchrones encore plus lisibles en supprimant la nécessité de chaîner des promesses avec **.then()**. Voici le même exemple que précédemment, mais en utilisant **async/await** :

```
async function sentUser(user) {
   try {
        const response = await fetch("https://api.exemple.com/
users", {
            method: "POST",
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify(user)
        });

        if (!response.ok) {
                        throw new Error("Erreur HTTP : " +
response.status);
        }

        const data = await response.json();
        console.log("Données reçues du serveur :", data);
    } catch (error) {
          console.error("Erreur lors de l'envoi de la requête
POST :", error);
    }
}

const user = {
    name: "Jean",
    age: 30,
    city: "Paris",
    country: "France"
};
```

```
sentUser(user);
```

Toujours spécifier les en-têtes de contenu
Lorsque vous travaillez avec des requêtes POST, PUT ou DELETE, assurez-vous de toujours spécifier les en-têtes corrects. Par exemple, si vous envoyez des données JSON, utilisez Content-Type: application/json. Si vous envoyez des données de formulaire, utilisez Content-Type: application/x-www-form-urlencoded.

Utiliser try/catch pour capturer les erreurs
Lors de l'utilisation de async/await, il est essentiel d'encapsuler les appels dans un bloc try/catch pour capturer et gérer les erreurs efficacement. Cela permet de garantir que toute erreur, qu'elle soit liée au réseau, à la réponse, ou à d'autres éléments, est bien gérée et n'arrête pas l'exécution de votre application de manière inattendue.

Traitement des réponses JSON : Comment manipuler des données JSON dans les réponses des API

Lorsque vous envoyez une requête à une API, la réponse du serveur est souvent sous forme de JSON. Les données reçues doivent être interprétées et manipulées pour être utilisées dans votre application. Voyons comment procéder.

Conversion de la réponse en JSON
Dans la Fetch API, après avoir reçu la réponse d'une requête HTTP, vous devez d'abord la convertir en JSON à l'aide de la méthode response.json(). Cela retourne une promesse qui se résout avec les données converties en un objet ou tableau JSON.

Voici un exemple simple montrant comment manipuler une réponse JSON après avoir envoyé une requête GET à une API :

```
    fetch("https://api.exemple.com/users")
        .then(response => {
            // Vérifie si la requête a réussi
            if (!response.ok) {
                        throw new Error("Erreur HTTP : " +
response.status);
            }
```

```
            // Convertit la réponse en JSON
            return response.json();
        })
        .then(data => {
            // Utilisation des données JSON
            console.log(data);
            // Exemple de manipulation des données
            data.forEach(user => {
                console.log("Nom:", user.name);
                console.log("Âge:", user.age);
            });
        })
        .catch(error => {
                 console.error("Erreur lors de la requête :",
error);
        });
```

Explication:

- **fetch()** : Envoie une requête HTTP GET à une API.
- **response.json()** : Cette méthode est utilisée pour convertir la réponse HTTP en objet JSON. Elle retourne une promesse qui se résout avec les données JSON.
- **Manipulation des données JSON** : Une fois la réponse convertie, nous avons accès aux données JSON. Dans cet exemple, nous parcourons les données pour afficher le nom et l'âge de chaque utilisateur.

Exemple de réponse JSON typique d'une API :

```
    [
        {
            "name": "Jean",
            "age": 30,
            "city": "Paris",
            "country": "France"
        },
        {
            "name": "Alice",
            "age": 25,
            "city": "Lyon",
```

```
            "country": "France"
        }
    ]
```

Dans cet exemple, **response.json()** convertit ce tableau JSON en une structure JavaScript manipulable, ce qui vous permet ensuite d'accéder et de manipuler facilement les informations pour les afficher, les traiter ou les stocker dans une autre variable.

Manipuler des données JSON complexes
Dans certains cas, les données JSON que vous recevez peuvent être plus complexes, avec des objets imbriqués ou des tableaux dans des objets. Voyons un exemple plus complexe.

Exemple avec JSON imbriqué :

```
{
    "users": [
        {
            "name": "Jean",
            "age": 30,
            "adress": {
                "city": "Paris",
                "postalCode": "75001"
            }
        },
        {
            "name": "Alice",
            "age": 25,
            "adress": {
                "city": "Lyon",
                "postalCode": "69001"
            }
        }
    ]
}
```

Exemple de traitement de cette réponse avec la Fetch API :

```
fetch("https://api.exemple.com/users")
    .then(response => {
```

```
            if (!response.ok) {
                        throw new Error("Erreur HTTP : " +
response.status);
            }
            return response.json();
        })
        .then(data => {
             // Accéder au tableau 'utilisateurs' dans l'objet
JSON
            const users = data.users;

            users.forEach(user => {
                console.log("Nom:", user.nom);
                console.log("Âge:", user.age);
                console.log("Ville:", user.adress.ville);
                                    console.log("Code Postal:",
user.adress.postalCode);
            });
        })
        .catch(error => {
                 console.error("Erreur lors de la requête :",
error);
        });
```

Explication :

- **Accès aux objets imbriqués** : L'exemple montre comment accéder aux propriétés imbriquées dans l'objet JSON. Ici, nous accédons à l'adresse de chaque utilisateur (user) et nous extrayons la ville (city) et le code postal (postalCode).

- **Boucle forEach()** : Nous utilisons forEach() pour itérer sur chaque utilisateur dans le tableau, affichant les informations pertinentes.

Utilisation avancée : Filtrer, transformer et stocker des données JSON

Les données JSON que vous recevez peuvent parfois nécessiter un traitement plus complexe, comme le filtrage ou la transformation avant leur affichage ou leur stockage.

Exemple : Filtrage des utilisateurs par âge

Supposons que vous souhaitiez extraire uniquement les utilisateurs de plus de 25 ans à partir d'un ensemble de données JSON.

```
fetch("https://api.exemple.com/users")
    .then(response => response.json())
    .then(data => {
        // Filtrer les utilisateurs ayant plus de 25 ans
              const usersFilters = data.users.filter(user =>
user.age > 25);

        usersFilters.forEach(user => {
            console.log("Nom:", user.name, "Âge:", user.age);
        });
    })
    .catch(error => {
        console.error("Erreur lors de la requête :", error);
    });
```

Explication :

- **filter()** : Nous utilisons la méthode filter() pour créer un nouveau tableau contenant uniquement les utilisateurs dont l'âge est supérieur à 25 ans.
- **Affichage des données filtrées** : Ensuite, nous affichons les informations des utilisateurs qui remplissent cette condition.

Conclusion

Dans ce chapitre, nous avons exploré en profondeur les requêtes HTTP en JavaScript, en nous concentrant sur les outils modernes que sont AJAX et la Fetch API. Nous avons vu comment ces technologies permettent à vos applications web de communiquer avec des serveurs de manière asynchrone, rendant possible le chargement de contenu dynamique sans avoir à recharger la page entière.

Nous avons également appris à :

- Récupérer des données avec les requêtes GET.
- Envoyer des données avec les requêtes POST.
- Traiter et manipuler des réponses JSON pour les intégrer dans vos applications.

La Fetch API, grâce à sa syntaxe plus moderne et à sa gestion des promesses, est devenue l'outil de choix pour les développeurs travaillant avec des requêtes HTTP. Sa simplicité et sa lisibilité, combinées avec des bonnes pratiques comme l'utilisation de async/await et la gestion des erreurs, en font un outil puissant pour développer des applications robustes.

En maîtrisant les concepts abordés dans ce chapitre, vous serez en mesure de créer des applications web plus dynamiques, interactives, et efficaces, capables de gérer des communications serveur complexes de manière fluide. N'hésitez pas à mettre ces connaissances en pratique dans vos projets pour bien vous familiariser avec ces outils indispensables du développement web moderne.

Chapitre 20

Les API Web

Dans l'univers du développement web moderne, les API Web sont devenues un outil incontournable pour permettre la communication entre différentes applications. API signifie Application Programming Interface (Interface de Programmation d'Applications), et une API Web permet à une application de communiquer avec un serveur pour envoyer et recevoir des données de manière asynchrone.

Les API Web sont au cœur des applications que nous utilisons tous les jours. Elles permettent à des services comme les réseaux sociaux, les applications de messagerie, les services de cartographie ou encore les plateformes de paiement d'interagir de manière transparente. En utilisant les API, les développeurs peuvent intégrer dans leurs propres projets des fonctionnalités et des services externes sans avoir à les développer depuis zéro.

Dans ce chapitre, nous allons explorer :

- Ce que sont les API Web, comment elles fonctionnent et pourquoi elles sont si importantes.
- Les différentes méthodes HTTP utilisées pour interagir avec les API, telles que GET, POST, PUT et DELETE.
- L'authentification et la gestion des clés API pour sécuriser les échanges avec des services externes.
- Comment intégrer les API Web dans vos applications JavaScript grâce à Fetch API ou encore Axios.

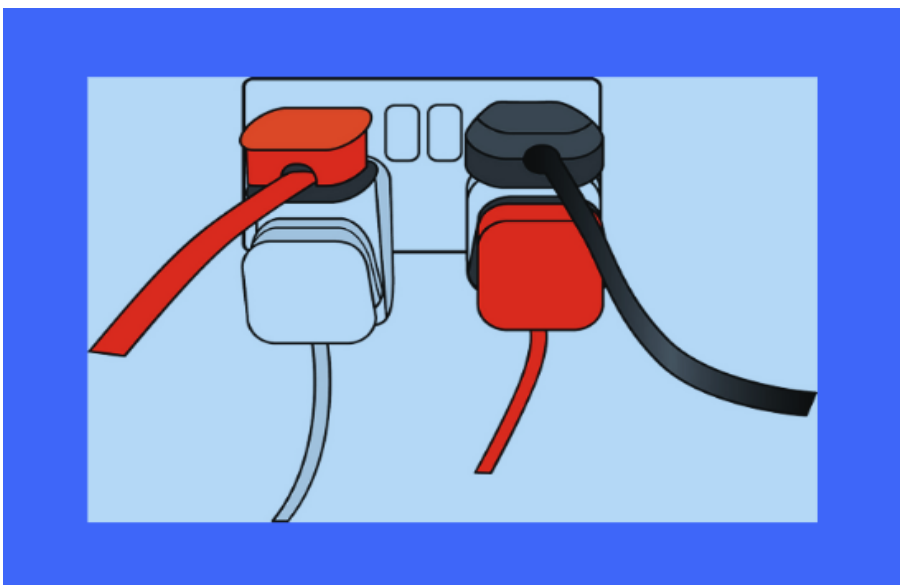
Les API Web vous permettent d'accéder à un monde de données et de services externes, transformant ainsi une simple application web en une plateforme riche et interactive. Que ce soit pour récupérer des informations depuis une base de données distante, accéder à des services tiers ou connecter votre application à d'autres services, maîtriser les API Web est essentiel pour tout développeur souhaitant créer des applications web modernes et performantes.

Qu'est-ce qu'une API Web ?

Une API Web (Application Programming Interface) est une interface de programmation d'application qui permet aux développeurs d'accéder à des fonctionnalités, des services ou des données d'une autre application, d'un service web, ou d'un serveur, sans avoir à comprendre ou à interagir directement avec le code complexe sous-jacent.

Les API fournissent une abstraction des opérations plus complexes, en offrant une syntaxe simplifiée et facile à utiliser. Elles permettent d'effectuer des tâches sophistiquées ou d'obtenir des informations de manière pratique, sans avoir à réinventer la roue.

Exemple illustratif
Imaginez que vous souhaitiez brancher des appareils électriques chez vous, comme un réfrigérateur, une télévision ou encore recharger votre téléphone portable. Plutôt que de brancher directement chaque appareil à la source d'alimentation (ce qui serait dangereux et complexe pour quelqu'un qui n'est pas électricien), vous utilisez des prises électriques pour connecter ces appareils de manière simple et sécurisée. Les API fonctionnent de la même manière : elles permettent de se connecter à des systèmes complexes (des services web, par exemple) via des points d'accès simples, comme des prises, sans avoir à comprendre tous les détails internes.



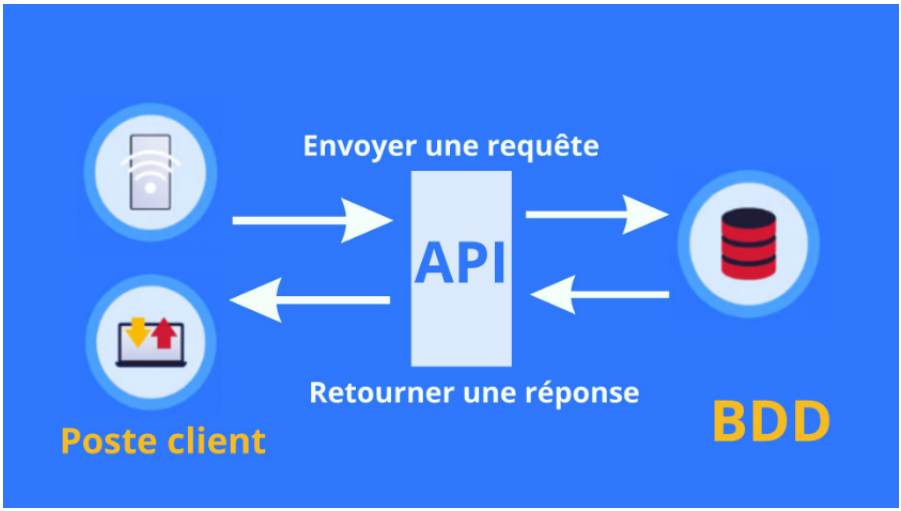
Exemple pratique : API de données météorologiques
Supposons que vous souhaitez afficher les conditions météorologiques en temps réel sur votre site web pour une ville spécifique. Il serait impossible de mettre en place tout un système pour collecter et mettre à jour ces données vous-même. Cependant, grâce à une **API** d'un service météorologique, vous pouvez récupérer facilement les données météorologiques à jour, sans effort. En quelques lignes de code, vous avez accès à ces informations fournies par un tiers spécialisé.

Exemple pratique : API Twitter
Prenons un autre exemple : vous souhaitez afficher vos derniers tweets sur votre site web. Il serait inefficace de récupérer manuellement ces tweets sur Twitter pour les afficher à jour sur votre site. Heureusement, **Twitter** propose une **API** qui permet de consulter automatiquement et afficher vos tweets récents. Cela simplifie considérablement la tâche et permet d'automatiser la récupération de ces informations.

En résumé, les API agissent comme des intermédiaires simples et accessibles, permettant d'utiliser des services externes sans devoir comprendre le fonctionnement interne des systèmes sous-jacents. Elles sont essentielles pour l'intégration de données ou de services dans des applications web modernes.

Comment fonctionnent les API ?

Pour bien illustrer le fonctionnement d'une API (Application Programming Interface) dans le contexte d'une interaction entre un poste client et une base de données (BDD), je vous propose le schéma explicatif suivant ci-dessous. Cet exemple est très pertinent pour expliquer comment fonctionne l'interaction des API dans un écosystème de services web. Que ce soit pour récupérer des données, comme des prévisions météo, ou pour interagir avec des plateformes comme Twitter ou YouTube, ce principe reste le même.



Explication détaillée du schéma :

- **Poste client :**

Le poste client est l'appareil utilisé par un utilisateur, cela peut être un smartphone, une tablette, ou un ordinateur. Ce poste est responsable de l'envoi de la requête à l'API. Dans ce contexte, l'utilisateur, via une application ou un site web, souhaite obtenir ou envoyer des informations.

- **Envoyer une requête :**

Lorsque l'utilisateur souhaite interagir avec une application, comme par exemple chercher des informations sur un produit ou envoyer un formulaire, une requête est envoyée depuis le poste client vers l'API. Cette requête contient les informations nécessaires pour demander une opération à l'API, comme une requête GET pour récupérer des données ou une requête POST pour envoyer des données.

- **API (Application Programming Interface) :**

L'API agit comme un intermédiaire entre le client et la base de données. Elle reçoit les requêtes du poste client, les traite, et ensuite, en fonction de la demande, interagit avec la base de données. Une fois que l'API reçoit la réponse de la BDD, elle formate les données sous une forme que le client peut comprendre (souvent au format JSON) et renvoie cette réponse au poste client. L'API simplifie et sécurise l'accès aux données tout en masquant les détails techniques de la base de données.

- **Base de données (BDD) :**

La base de données est l'endroit où les informations sont stockées. Cela peut être des données sur des utilisateurs, des produits, des

commandes, etc. Lorsque l'API interroge la base de données, elle effectue une opération comme une récupération d'informations (requête SELECT), une mise à jour (requête UPDATE), une insertion (requête INSERT), ou une suppression (requête DELETE).

- **Retourner une réponse :**

Après que l'API a traité la requête et obtenu les informations de la base de données, elle renvoie une réponse au client. Cette réponse contient les données demandées ou une confirmation que l'opération a été effectuée avec succès (par exemple, un utilisateur a été ajouté ou mis à jour dans la base de données). Cette réponse est ensuite affichée à l'utilisateur sur l'interface du poste client.

- **Synthèse du processus :**

L'utilisateur interagit avec une application sur son poste client.
Le poste client envoie une requête à l'API pour obtenir ou modifier des données.
L'API reçoit la requête et la transmet à la base de données.
La base de données traite la requête et renvoie les résultats à l'API.
L'API renvoie ensuite une réponse au poste client, qui l'affiche à l'utilisateur.
Ce schéma montre à quel point l'API simplifie l'interaction entre un utilisateur et un système complexe (comme une base de données) en offrant une interface simple d'utilisation et en masquant les détails techniques sous-jacents.

Les API JavaScript côté client

Les développeurs JavaScript ont accès à un large éventail d'API. Il existe des API Web intégrées dans tous les navigateurs, ainsi que des API tierces fournies par des services externes et non intégrées directement dans les navigateurs.

API Web (intégrées dans les navigateurs) :
Les navigateurs modernes intègrent de nombreuses API qui permettent de simplifier le développement d'applications web. Voici quelques exemples d'API populaires :

- **API DOM** : Le DOM, que nous avons déjà étudié, est une API intégrée permettant de manipuler la structure et le contenu d'un document HTML ou XML.
- **API XMLHttpRequest** : Cette API permet de faire des requêtes HTTP asynchrones depuis JavaScript, pour mettre à jour une page sans la recharger entièrement.

- **API Geolocation** : Elle permet d'accéder à la position géographique du client.
- **API Web Storage** : Elle permet de stocker des données localement dans le navigateur.
- **API Canvas** : Elle permet de dessiner des graphismes, des images, et des animations avec JavaScript.
- **API Web Audio** : Elle permet de créer et manipuler des sons à l'aide de JavaScript.
- **API Web Workers** : Elle permet d'exécuter des scripts JavaScript en arrière-plan pour libérer le thread principal.
- **File API** : Elle permet d'accéder aux fichiers sélectionnés par l'utilisateur via un formulaire.

Ces API sont indispensables pour créer des applications web interactives et riches en fonctionnalités.

Exemple de l'API de géolocalisation

Supposons que vous souhaitiez obtenir la position géographique d'un utilisateur. Avant de commencer, notez que cette position ne peut être récupérée que si l'utilisateur accepte de la partager.

Création de l'interface HTML

Dans votre fichier index.html, nous allons créer deux éléments <h2> pour afficher les coordonnées géographiques :

```
    <!DOCTYPE html>
    <html lang="fr">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
            <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <title>JavaScript Asynchrone</title>
    </head>

    <body>
        <h1>Coordonnées géographiques</h1>
        <h2 id="latitude"></h2>
        <h2 id="longitude"></h2>

        <script src="my_script.js"></script>
```

```
    </body>

    </html>
```

Script JavaScript pour obtenir la géolocalisation

Dans my_script.js, nous allons écrire le code pour vérifier si le navigateur prend en charge l'API de géolocalisation et, si l'utilisateur accepte, afficher ses coordonnées :

```
      // Vérifier si le navigateur prend en charge l'API de
géolocalisation
    if (navigator.geolocation) {
        // Demander la permission de géolocalisation

navigator.geolocation.getCurrentPosition(showPosition);
    } else {
             // Afficher un message d'erreur si l'API de
géolocalisation n'est pas disponible
         console.log("Désolé, votre navigateur ne prend pas en
charge l'API de géolocalisation");
    }

    function showPosition(position) {
            // Récupérer la latitude et la longitude de la
position actuelle
        const latitude = position.coords.latitude;
        const longitude = position.coords.longitude;
        // Afficher les coordonnées dans la console
            console.log(`Latitude : ${latitude}, Longitude :
${longitude}`);
    }
```

Explication:

- **Vérification de la compatibilité** : On commence par vérifier si le navigateur supporte l'API de géolocalisation en utilisant **navigator.geolocation**. Si l'API est supportée, une demande de permission est envoyée à l'utilisateur via la méthode **getCurrentPosition().** Cette

méthode prend en argument une fonction de rappel, ici **showPosition**, qui sera appelée si l'utilisateur autorise la géolocalisation.

- **Fonction showPosition()** : Cette fonction prend un paramètre position qui contient les informations géographiques de l'utilisateur. Nous accédons à la latitude et à la longitude via **position.coords.latitude** et **position.coords.longitude** et les affichons dans la console.

Lorsque vous exécutez ce script et si vous avez autorisé la localisation de votre position, vous obtiendrez, dans la console de votre navigateur, l'affichage de la latitude et de la longitude de votre position.

Affichage des coordonnées dans la page web
Plutôt que d'afficher les coordonnées uniquement dans la console, nous allons les afficher directement sur la page.

Modifions la fonction **showPosition()** pour mettre à jour les éléments <h2> créés précédemment dans index.html :

```
  function showPosition(position) {
            // Récupérer la latitude et la longitude de la position actuelle
        const latitude = position.coords.latitude;
        const longitude = position.coords.longitude;

        // Mettre à jour le contenu des éléments HTML
                                const latitudeElement = document.getElementById("latitude");
                                const longitudeElement = document.getElementById("longitude");
        latitudeElement.innerText = `Latitude : ${latitude}`;
         longitudeElement.innerText = `Longitude : ${longitude}`;
    }
```

Explication :

- **Sélection des éléments HTML** : Nous utilisons **document.getElementById()** pour sélectionner les éléments avec les IDs latitude et longitude.

- **Mise à jour du contenu** : Ensuite, nous utilisons **innerText** pour modifier le texte affiché dans ces éléments et y insérer les coordonnées récupérées.

Les API intégrées dans les navigateurs, comme l'API de géolocalisation, facilitent grandement l'ajout de fonctionnalités interactives et puissantes dans les applications web. Elles permettent de capter des informations en temps réel, d'interagir avec le matériel (comme la localisation), et de rendre vos applications plus dynamiques, tout en restant faciles à implémenter.

Dans cet exemple, nous avons vu comment utiliser la géolocalisation pour obtenir et afficher les coordonnées d'un utilisateur dans une page web, mais cette API peut aussi être utilisée pour bien d'autres applications, comme la génération de cartes interactives.

Les API tierces (fournies par des services externes)

Contrairement aux API intégrées dans les navigateurs, les API tierces sont fournies par des services externes et ne sont pas directement disponibles dans votre navigateur. Pour les utiliser, il est souvent nécessaire de télécharger des bibliothèques ou d'importer des scripts à partir du web.

Il existe une grande variété d'API tierces, certaines sont gratuites tandis que d'autres sont payantes. Ces API permettent aux développeurs d'ajouter des fonctionnalités complexes et avancées à leurs applications sans avoir à tout développer à partir de zéro. Voici quelques exemples d'API tierces populaires :

- **Google Maps API** : Permet d'intégrer des cartes interactives et des fonctionnalités de géolocalisation dans vos applications web.
- **Twitter API** : Permet d'afficher vos derniers tweets sur un site web ou d'accéder aux données de Twitter, comme les tweets, les tendances, et les profils.
- **Stripe API** : Utilisée pour accepter des paiements en ligne de manière sécurisée.
- **Twilio API** : Permet d'ajouter des fonctionnalités de communication, telles que l'envoi de messages texte ou la réalisation d'appels vocaux, directement dans vos applications.
- **OpenWeatherMap API** : Fournit des données météorologiques en temps réel pour n'importe quelle ville dans le monde.

- **GitHub API** : Permet d'accéder aux informations des dépôts, profils, et activités des utilisateurs de GitHub.
- **Spotify API** : Permet d'accéder à la bibliothèque musicale de Spotify et de créer des applications autour de la musique.

Ces API tierces sont très utiles car elles permettent aux développeurs d'ajouter rapidement des fonctionnalités avancées sans avoir à recréer des services complexes. Cela permet de se concentrer sur la création d'une application tout en s'appuyant sur des outils robustes et fiables fournis par ces services externes.

Comment utiliser une API tierce ?
Pour intégrer une API dans votre application, il est essentiel de consulter la documentation officielle de l'API. Cette documentation contient des informations détaillées sur :

- **Les méthodes disponibles** : actions que vous pouvez effectuer avec l'API.
- **Les paramètres nécessaires** : données que vous devez envoyer avec vos requêtes.
- **Les formats de réponse** : la manière dont l'API renvoie les données, souvent en JSON ou XML.

Chaque API possède ses propres caractéristiques et ses propres fonctionnalités, c'est pourquoi il est crucial de bien comprendre son fonctionnement en consultant la documentation pour tirer le meilleur parti de ses possibilités.

Conclusion

Les API Web, qu'elles soient intégrées dans les navigateurs ou fournies par des services externes, sont des outils puissants qui permettent d'enrichir les fonctionnalités des applications web. Grâce à elles, vous pouvez intégrer des cartes interactives, afficher des données en temps réel, faciliter des paiements en ligne ou encore ajouter des fonctionnalités de communication avancées sans avoir à écrire tout le code sous-jacent.

Les API tierces, en particulier, ouvrent un monde de possibilités en vous permettant d'exploiter des services existants, fiables et souvent performants, pour créer des applications riches en fonctionnalités, tout en vous concentrant sur l'expérience utilisateur et le développement rapide.

Dans le prochain chapitre, nous mettrons en pratique ce que nous avons appris avec un projet d'application de météo géolocalisé. Ce projet nous permettra d'utiliser des API pour récupérer les informations météorologiques en fonction de la localisation de l'utilisateur et d'afficher les résultats en temps réel. Nous découvrirons également comment combiner les API de géolocalisation et une API météo pour créer une application fonctionnelle et interactive.

Ce sera une excellente occasion de voir comment les API peuvent transformer un concept en application web pratique et utile.

Chapitre 21

Projet d’application météo géolocalisée

L'objectif de ce projet était de créer une application web de météorologie interactive qui afficherait en direct les conditions météorologiques actuelles pour n'importe quel emplacement sélectionné. Afin de réaliser cela, nous avons choisi d'utiliser gratuitement l'API **OpenWeatherMap**, qui nous permettrait d'extraire les informations météorologiques nécessaires pour une ville spécifiée. Nous avons construit cette application à l'aide des langages HTML, CSS et JavaScript, ce qui nous a permis de générer une interface conviviale répondant en temps réel aux requêtes des utilisateurs. Bien que la tâche se soit avérée quelque peu ardue en raison de la nature dispersée des données, nous avons néanmoins réussi à créer un outil fonctionnel grâce à notre persévérance et à nos compétences conjointes en développement web.

Étape 1 : Obtenir une clé API OpenWeatherMap

Pour pouvoir utiliser l'API OpenWeatherMap, vous devez créer un compte gratuit afin de recevoir une clé API.

- Rendez-vous sur le site : https://openweathermap.org
- Créez un compte et validez votre adresse e-mail. Connectez-vous si ce n'est pas déjà fait.
- Dans le menu, sélectionnez « API ».
- Choisissez « Current Weather Data ». À l'heure où nous écrivons, le site propose 5 formules d'abonnement (1 gratuite et 4 payantes).
- Pour cette démonstration, nous choisirons l'option gratuite. Cliquez sur « Get API Key » dans la première colonne.

La version gratuite vous permet d'effectuer jusqu'à 60 appels par minute et un maximum de 1 million d'appels par mois, avec des prévisions toutes les 3 heures sur une période de 5 jours. C'est largement suffisant pour notre application.

Une fois votre compte créé, sélectionnez votre nom d'utilisateur (en haut à droite), puis cliquez sur « My API keys ». Vous pouvez cliquer sur « Generate » pour obtenir une nouvelle clé, ou utiliser la clé générée par défaut.

N'oubliez pas de noter votre clé API, nous en aurons besoin dans le code ultérieurement.

Architecture de l'application

Les informations météorologiques que nous souhaitons afficher dans notre application incluent :

- La température actuelle,
- La température maximale et minimale,
- Une description du climat,
- Le taux d'humidité,
- La vitesse du vent,
- La visibilité.

Notre application sera évolutive et pourra récupérer d'autres données si nécessaire, car l'API OpenWeatherMap fournit une multitude d'informations météorologiques en temps réel.

Nous commencerons par créer la structure de notre page d'accueil : meteo.html. Toutes les informations météorologiques seront récupérées dynamiquement via l'API OpenWeatherMap, mais pour l'instant, nous allons mettre en place un design avec des valeurs fictives.

Création du template

Commencez par créer un dossier que vous pouvez nommer "meteo". À

l'intérieur de ce dossier, créez deux sous-dossiers :

- css : ce dossier contiendra tous les fichiers de style CSS.
- img_meteo : ce dossier sera dédié aux images nécessaires pour l'interface de l'application météo. Vous trouverez toutes les images utilisées dans ce projet dans le dossier de code source à télécharger (lien à la fin du livre).

Ensuite, dans le répertoire principal, créez un fichier HTML nommé meteo.html, qui sera la page principale de notre application.

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
          <meta   name="viewport"   content="width=device-width,
initial-scale=1.0">
    <link rel="stylesheet" href="css/style.css">
    <title>Weather App</title>
</head>

<body>
    <div class="container">
        <div class="formSearch">
            <input type="text" placeholder="Nom de la ville">
              <img src="img_meteo/search-icon.png" alt="Search
Icon">
        </div>
        <div class="weather">
             <img src="img_meteo/sun-icon.png" class="weather-
icon" alt="Weather Icon">
            <h2 class="city">Paris</h2>
            <h1 class="temperature">30°C</h1>
            <p id="description">Ensoleillé</p>
            <p id="maxmin">Min: 17°C, Max: 31°C</p>

            <div class="infos">
                <div class="column">
                        <img src="img_meteo/humidity-icon.png"
alt="Humidity Icon">
```

```
                    <div>
                        <p>Humidité</p>
                        <p class="humidity">70%</p>
                    </div>
                </div>

                <div class="column">
                            <img src="img_meteo/wind-icon.png"
alt="Wind Icon">
                    <div>
                        <p>Vent</p>
                        <p class="wind">55 km/h</p>
                    </div>
                </div>

                <div class="column">
                       <img src="img_meteo/visibility-icon.png"
alt="Visibility Icon">
                    <div>
                        <p>Visibilité</p>
                        <p class="visibility">300 m</p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</body>

</html>
```

Feuille de style style.css
Ensuite, créez un fichier style.css dans le dossier css pour styliser notre application. Voici le contenu de cette feuille de style :

```
    .container {
        background-color: #f2f2f2;
        padding: 30px;
        width: 100%;
         /* Ajout pour que le container prenne toute la largeur
de la fenêtre */
```

```
        max-width: 400px;
        /* Limiter la largeur pour qu'elle ne dépasse pas */
        border-radius: 15px;
        text-align: center;
        margin: auto;
        font-family: Arial, Helvetica, sans-serif;
    }

    .formSearch {
        width: 100%;
        margin-bottom: 10px;
        position: relative;
        box-sizing: border-box;
          /* Assure que le padding et la largeur sont pris en
compte correctement */
    }

    .formSearch input {
        width: 100%;
            /* Utilisation de toute la largeur disponible du
parent */
        padding: 10px 40px 10px 20px;
        /* Espace pour l'icône à droite */
        margin-right: 0;
        border: none;
        border-radius: 30px;
        font-size: 17px;
        box-shadow: 2px 2px 5px rgba(0, 0, 0, 0.1);
        box-sizing: border-box;
          /* S'assure que le padding n'affecte pas la largeur
totale */
    }

    .formSearch img {
        position: absolute;
        right: 10px;
         /* Ajuster la position depuis le bord droit de l'input
*/
        top: 50%;
```

```
        transform: translateY(-50%);
            /* Centrage vertical de l'icône dans le champ de
recherche */
        width: 20px;
        height: 20px;
        pointer-events: none;
        /* Évite que l'icône interfère avec la saisie */
    }

    .weather-icon {
        width: 130px;
        margin-top: 25px;
    }

    .city {
        font-size: 25px;
        font-weight: 400;
    }

    .temperature {
        font-size: 40px;
        font-weight: 500;
    }

    .infos {
        display: flex;
        align-items: center;
        justify-content: space-between;
        padding: 5px 15px;
        margin-top: 40px;
    }

    .humidity,
    .wind,
    .visibility {
        font-size: 22px;
        margin-top: -10px;
    }
```

Aperçu de l'application

L'interface météorologique de base affiche pour l'instant des valeurs factices. Le bouton de recherche trône à droite du champ d'entrée pour une navigation plus intuitive.

Avant d'intégrer la fonction de recherche et de collecte des données météo en temps réel via l'API OpenWeatherMap dans le code JavaScript, nous avons terminé la conception graphique de l'application. Le design final apporte plus de clarté et de convivialité à l'utilisateur. Nous avons hâte de connecter cette interface épurée à une source de données réelles.

Utilisation de l'API OpenWeatherMap

Créons un fichier JavaScript nommé app.js et ajoutons un lien vers ce script dans la page meteo.html.

Nous allons commencer par enregistrer notre clé API dans une variable nommée api_key.

```
// On va enregistrer notre clé dans une variable api_key
const api_key = 'c3fe3795961723d1b93a169c20b80bd5';
// On crée une variable pour stocker le nom de la ville
const city = 'Paris';
```

Bien que l'utilisation des coordonnées géographiques soit recommandée pour une précision accrue, il est possible d'obtenir des prévisions météorologiques en fournissant uniquement le nom de la localité souhaitée. Cependant, le géocodeur intégré est devenu obsolète au fil du temps.

Dans le cadre de cet exemple simple, nous nous contenterons de la dénomination de la ville. Un projet avancé gagnerait à mettre en œuvre les coordonnées, ce qui assurerait une application météorologique à la fois plus raffinée et plus performante.

Création de la fonction pour récupérer les données météo

Commençons par créer une fonction asynchrone appelée getWeather(). Nous allons y spécifier en premier lieu l'URL de l'API sous la forme d'une variable.

Puis, nous aurons recours à la méthode fetch pour interroger l'API OpenWeatherMap et ainsi récupérer les prévisions désirées. Cette requête s'effectuera via la méthode GET afin d'obtenir les données souhaitées.

```
    async function getWeather() {
            // On crée une variable url pour stocker l'URL de
l'API
          const url = `https://api.openweathermap.org/data/2.5/
weather?q=${city}&units=metric&appid=${api_key}`;

        const response = await fetch(url);
        let data = await response.json();
        console.log(data);
    }
    // Appel de la fonction getWeather()
    getWeather();
```

Lorsque les données météorologiques sont récupérées avec succès à partir de l'API, elles s'affichent de manière détaillée dans la console grâce à la fonction console.log().

Résultats

En exécutant le code, un objet JSON contenant des informations précises sur les conditions atmosphériques apparait dans la console. Il est possible d'inspecter en profondeur les détails météo en développant de façon interactive la structure de données imbriquée.

Le site web officiel documentant l'API météo propose également une explication détaillée de la signification de chaque élément renvoyé.

Récupération des informations météorologiques

Par la suite, nous extrayons des valeurs météorologiques majeures comme la température, l'humidité et la vitesse du vent pour les afficher de manière synthétique dans la console.

Voici un exemple complet du code pour récupérer ces informations :

```
async function getWeather() {
    // On crée une variable url pour stocker l'URL de l'API
        const url = `https://api.openweathermap.org/data/2.5/
weather?q=${city}&units=metric&appid=${api_key}`;

    const response = await fetch(url);
    let data = await response.json();

    // Récupérer les valeurs des informations météorologiques
    const cityName = data.name;
    const temperature = data.main.temp;
    const tempMax = data.main.temp_max;
    const tempMin = data.main.temp_min;
    const humidity = data.main.humidity;
    const windSpeed = data.wind.speed;
    const visibility = data.visibility;
    const weatherDescription = data.weather[0].description;
```

```
    console.log(cityName);
    console.log(temperature);
    console.log(tempMax);
    console.log(tempMin);
    console.log(humidity);
    console.log(windSpeed);
    console.log(visibility);
    console.log(weatherDescription);
}

// Appel de la fonction getWeather()
getWeather();
```

Résultat

En exécutant ce code, vous obtiendrez les valeurs météorologiques à afficher dans la console. Par exemple, la température s'affichera en degrés Kelvin. Il suffit d'ajouter le paramètre "units=metric" dans l'URL pour la convertir en degrés Celsius.

Voici l'URL modifiée :

```
        const url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&units=metric&appid=${api_key}`;
```

Désormais, les températures s'afficheront en degrés Celsius.

L'humidité, exprimée en pourcentage, ainsi que la vitesse du vent, mesurée en kilomètres par heure, seront également récupérées et affichées.

```
Paris
16.72
17.36
15.34
64
4.12
10000
few clouds
```

Affichage des informations météorologiques

Dans le fichier app.js, remplaçons le code des console.log par le code suivant afin d'afficher les informations météorologiques directement dans la page meteo.html :

```
// Mise à jour des éléments HTML avec les valeurs récupérées
    document.querySelector(".city").innerHTML = cityName;
        document.querySelector(".temperature").innerHTML = `
${temperature} °C`;
         document.querySelector("#maxmin").innerHTML = `Min:
${tempMin}°C, Max: ${tempMax}°C`;
          document.querySelector("#description").innerHTML =
weatherDescription;
          document.querySelector(".humidity").innerHTML = `
${humidity} %`;
     document.querySelector(".wind").innerHTML = `${windSpeed}
Km/h`;
         document.querySelector(".visibility").innerHTML = `
${visibility} m
```

Le code de appa.js devient :

```
// On va enregistrer notre clé dans une variable api_key
const api_key = 'c3fe3795961723d1b93a169c20b80bd5';
// On crée une variable pour stocker le nom de la ville
const city = 'Paris';

async function getWeather() {
    // On crée une variable url pour stocker l'URL de l'API
        const url = `https://api.openweathermap.org/data/2.5/
weather?q=${city}&units=metric&appid=${api_key}`;
```

```
    const response = await fetch(url);
    let data = await response.json();

    // Récupérer les valeurs des informations météorologiques
    const cityName = data.name;
```

```
    const temperature = data.main.temp;
    const tempMax = data.main.temp_max;
    const tempMin = data.main.temp_min;
    const humidity = data.main.humidity;
    const windSpeed = data.wind.speed;
    const visibility = data.visibility;
    const weatherDescription = data.weather[0].description;

     // Afficher les informations météorologiques dans la page
HTML
// Mise à jour des éléments HTML avec les valeurs récupérées
    document.querySelector(".city").innerHTML = cityName;
         document.querySelector(".temperature").innerHTML  =  `
${temperature} °C`;
          document.querySelector("#maxmin").innerHTML  =  `Min:
${tempMin}°C, Max: ${tempMax}°C`;
           document.querySelector("#description").innerHTML  =
weatherDescription;
          document.querySelector(".humidity").innerHTML  =  `
${humidity} %`;
     document.querySelector(".wind").innerHTML = `${windSpeed}
Km/h`;
          document.querySelector(".visibility").innerHTML  =  `
${visibility} m
}

// Appel de la fonction getWeather()
getWeather();
```

Si vous exécutez ce code, vous obtiendrez la météo de Paris :

Dynamisation de l'application

À ce stade, nous avions réussi à récupérer et afficher les informations météorologiques dans la page meteo.html. Cependant, le nom de la ville était actuellement fixé dans le code, alors que nous souhaitions rendre l'application plus dynamique pour qu'elle puisse afficher les données météorologiques de la ville saisie par l'utilisateur via le formulaire.

Afin d'atteindre cet objectif, nous avons tout d'abord dû localiser l'élément de champ du formulaire contenant la classe inputCity. Une fois identifié, il nous fallait ensuite relier la valeur entrée à notre requête d'API météorologique avant d'actualiser la page avec les nouvelles données. Bien que la tâche semblait simple, des défis techniques sont rapidement apparus concernant la gestion des événements et l'envoi de requêtes asynchrones. On va d'abord commencer par récupérer du formulaire correspondant au champ de texte et ayant comme classe inputCity.

```
const city = document.querySelector(".inputCity");
```

Il est important de mettre à jour le code pour permettre à l'utilisateur de saisir le nom de la ville de son choix. Tout d'abord, ajoutons la

classe "inputCity" à l'élément input du formulaire météo afin de pouvoir récupérer la valeur saisie. Ensuite, modifions la fonction de récupération des données météo pour qu'elle accepte un paramètre "location" qui sera utilisé à la place du nom de ville fixe dans l'URL. Lors de l'appel de la fonction, transmettons simplement le contenu du champ input comme argument, de sorte que les prévisions correspondront automatiquement à la ville renseignée par l'utilisateur.

```
async function getWeather(location) {
    // On crée une variable url pour stocker l'URL de l'API
        const url = `https://api.openweathermap.org/data/2.5/
weather?q=${location}&units=metric&appid=${api_key}`;

    // Le reste du code reste inchangé
}
```

Nous créons ensuite un événement sur le champ de saisie qui déclenche l'appel de la fonction getWeather() lorsque l'utilisateur tape le nom de la ville :

```
city.addEventListener("input", () => {
       //récupérer le nom de la ville (la valeur du champ
inputVille)
    const cityName = city.value;
    //Appel de la focntion getMeteo(location)
    getWeather(cityName);
});
```

À ce stade, voici le code complet de app.js :

```
// On va enregistrer notre clé dans une variable api_key
const api_key = 'c3fe3795961723d1b93a169c20b80bd5';
const city = document.querySelector(".inputCity");

// Fonction pour obtenir les informations météo
async function getWeather(location) {
     // On crée une variable url pour stocker l'URL de l'API
avec le nom de la ville
        const url = `https://api.openweathermap.org/data/2.5/
weather?q=${location}&units=metric&appid=${api_key}`;
```

```
    const response = await fetch(url);
    const data = await response.json();
```

```
    // Récupérer les valeurs des informations météorologiques
    const cityName = data.name;
    const temperature = data.main.temp;
    const tempMax = data.main.temp_max;
    const tempMin = data.main.temp_min;
    const humidity = data.main.humidity;
    const windSpeed = data.wind.speed;
    const visibility = data.visibility;
    const weatherDescription = data.weather[0].description;

        // Mise à jour des éléments HTML avec les valeurs
récupérées
    document.querySelector(".city").innerHTML = cityName;
         document.querySelector(".temperature").innerHTML = `
${temperature} °C`;
          document.querySelector("#maxmin").innerHTML = `Min:
${tempMin}°C, Max: ${tempMax}°C`;
           document.querySelector("#description").innerHTML =
weatherDescription;
           document.querySelector(".humidity").innerHTML = `
${humidity} %`;
      document.querySelector(".wind").innerHTML = `${windSpeed}
Km/h`;
          document.querySelector(".visibility").innerHTML = `
${visibility} m`;

}

city.addEventListener("input", () => {
        //récupérer le nom de la ville (la valeur du champ
inputVille)
    const cityName = city.value;
    //Appel de la focntion getMeteo(location)
    getWeather(cityName);
});
```

Modification de la langue :

Pour modifier la langue des données météorologiques et les afficher en français, il suffit d'ajouter le paramètre &lang=fr dans l'URL de la requête API. Cela permet d'obtenir les descriptions météorologiques en langue française.

Ainsi, la variable url devient :

```
        const  url  =  `https://api.openweathermap.org/data/2.5/
weather?q=${location}&units=metric&lang=fr&appid=${api_key}`;
```

Si vous testez votre application, vous remarquez que la description est affichée en langue française :

Jusqu'à présent, l'image climatique affichée dans notre application était statique et identique pour tous les lieux. Cependant, l'image devrait refléter les conditions météorologiques actuelles en fonction des données météorologiques en temps réel. Nous allons voir comment faire en sorte que l'application récupère automatiquement l'icône correspondant à la description du climat.

Si nous utilisions console.log(données) et examinions le résultat dans la console, nous obtiendrions un aperçu contenant diverses informations sur les conditions météorologiques actuelles. De plus, des phrases plus courtes sont entrecoupées de phrases plus longues et complexes.

Dans l'objet weather, à l'indice 0, nous avons deux clés importantes :

- La clé description, qui contient une description textuelle du climat.
- La clé icon, qui contient le nom de l'icône à utiliser pour représenter le climat.

Nous pouvons donc récupérer cette icône via la clé icon en stockant sa valeur dans une variable icon :

```
const icon = data.weather[0].icon;
```

D'après la documentation de l'API OpenWeatherMap, l'URL pour récupérer les icônes est de la forme suivante :

```
const iconUrl = `https://api.openweathermap.org/img/w/image.png`;
```

Pour obtenir l'icône correspondant à la météo actuelle, nous devons remplacer image.png par la valeur de la variable icon :

```
const iconUrl = `https://api.openweathermap.org/img/w/${icon}.png`;
```

Ensuite, nous devons modifier la source (src) de l'image HTML dans laquelle l'icône météorologique est affichée. Nous utilisons

querySelector pour sélectionner l'élément img et nous mettons à jour l'attribut src avec l'URL de l'icône :

```
document.querySelector(".meteo-icone").src = iconUrl;
```

Il ne reste plus qu'à tester l'application pour vérifier que l'image du climat change dynamiquement en fonction des conditions météorologiques actuelles de la ville saisie par l'utilisateur.

Voici le code complet de meteo.html :

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
          <meta   name="viewport"   content="width=device-width,
initial-scale=1.0">
    <link rel="stylesheet" href="css/style.css">
    <title>Weather App</title>
</head>

<body>
    <div class="container">
        <div class="formSearch">
             <!-- Classe inputVille pour capturer le nom de la
ville -->
                        <input  type="text"  class="inputCity"
placeholder="Nom de la ville">
            <img src="img_meteo/search-icon.png">
        </div>
        <div class="weather">
             <img src="img_meteo/sun-icon.png" class="weather-
icon" alt="Weather Icon">
            <!-- Nom de la ville mis à jour dynamiquement -->
            <h2 class="city">Nom de la ville</h2>
            <!-- Température mis à jour dynamiquement -->
            <h1 class="temperature">30°C</h1>
             <!-- Description météo mis à jour dynamiquement --
>
```

```
            <p id="description">Ensoleillé</p>
             <!-- Température min/max mis à jour dynamiquement
-->
            <p id="maxmin">Min: 17°C, Max: 31°C</p>

            <div class="infos">
                <div class="column">
                        <img src="img_meteo/humidity-icon.png"
alt="Humidity Icon">
                    <div>
                        <p>Humidité</p>
                              <!-- Taux d'humidité mis à jour
dynamiquement -->
                        <p class="humidity">70%</p>
                    </div>
                </div>

                <div class="column">
                            <img src="img_meteo/wind-icon.png"
alt="Wind Icon">
                    <div>
                        <p>Vent</p>
                              <!-- Vitesse du vent mis à jour
dynamiquement -->
                        <p class="wind">55 km/h</p>
                    </div>
                </div>

                <div class="column">
                       <img src="img_meteo/visibility-icon.png"
alt="Visibility Icon">
                    <div>
                        <p>Visibilité</p>
                                  <!-- Visibilité mis à jour
dynamiquement -->
                        <p class="visibility">300 m</p>
                    </div>
                </div>
            </div>
```

```
        </div>
    </div>
    <!-- Lien vers le fichier app.js -->
    <script src="app.js"></script>
</body>

</html>
```

Voici le code complet de app.js :

```
// On va enregistrer notre clé dans une variable api_key
const api_key = 'c3fe3795961723d1b93a169c20b80bd5';
const city = document.querySelector(".inputCity");

// Fonction pour obtenir les informations météo
async function getWeather(location) {
     // On crée une variable url pour stocker l'URL de l'API
avec le nom de la ville

       const url = `https://api.openweathermap.org/data/2.5/
weather?q=${location}&units=metric&lang=fr&appid=${api_key}`;

    const response = await fetch(url);
    const data = await response.json();
```

```
    // Récupérer les valeurs des informations météorologiques
    const cityName = data.name;
    const temperature = data.main.temp;
    const tempMax = data.main.temp_max;
    const tempMin = data.main.temp_min;
    const humidity = data.main.humidity;
    const windSpeed = data.wind.speed;
    const visibility = data.visibility;
    const weatherDescription = data.weather[0].description;
    const icon = data.weather[0].icon;
       const iconUrl = `https://api.openweathermap.org/img/w/
${icon}.png`;

       // Mise à jour des éléments HTML avec les valeurs
```

```
récupérées
    document.querySelector(".city").innerHTML = cityName;
        document.querySelector(".temperature").innerHTML = `
${temperature} °C`;
         document.querySelector("#maxmin").innerHTML = `Min:
${tempMin}°C, Max: ${tempMax}°C`;
          document.querySelector("#description").innerHTML =
weatherDescription;
          document.querySelector(".humidity").innerHTML = `
${humidity} %`;
     document.querySelector(".wind").innerHTML = `${windSpeed}
Km/h`;
         document.querySelector(".visibility").innerHTML = `
${visibility} m`;
    document.querySelector(".weather-icon").src = iconUrl;

}

city.addEventListener("input", () => {
      //récupérer le nom de la ville (la valeur du champ
inputVille)
    const cityName = city.value;
    //Appel de la focntion getMeteo(location)
    getWeather(cityName);
});
```

Conclusion

Ce projet impliquait le développement d'une application météorologique en temps réel permettant aux utilisateurs de consulter les conditions actuelles dans n'importe quelle ville. L'application extrait les données météorologiques depuis l'API **OpenWeatherMap**, affichant des éléments comme la température, l'humidité, la vitesse du vent et la visibilité ainsi qu'une icône représentant le temps (ensoleillé, nuageux et ainsi de suite).
Les utilisateurs peuvent saisir le nom d'une ville dans un champ de texte, ce qui met à jour dynamiquement les informations affichées en conséquence. Grâce à la récupération automatique de la description et

de l'icône météorologique via l'API, l'application offre une expérience interactive et actualisée en temps réel.

Principaux aspects techniques:

- L'API OpenWeatherMap fut utilisée pour obtenir les données météorologiques via une clé d'API enregistrée.
- JavaScript asynchrone: async/await a géré les appels réseau vers l'API et l'extraction des données.
- Mise à jour dynamique du DOM: Les informations météo sont directement injectées dans la page HTML via JavaScript pour un affichage en temps réel.
- Icône météorologique : L'application récupère et affiche une icône représentant les conditions de la ville saisie.

Potentielles améliorations:

- Recherche avancée : Améliorer la fonction de recherche en ajoutant des suggestions automatiques lors de la saisie du nom d'une ville (via une API de géocodage par exemple).
- Prévisions multiples : Actuellement, l'application n'affiche que les conditions actuelles. Il serait possible d'ajouter la fonctionnalité d'afficher les prévisions sur plusieurs jours.
- Localisation automatique: Ajouter une fonctionnalité pour détecter automatiquement la position géographique de l'utilisateur et afficher les conditions météo de sa ville actuelle sans saisie manuelle.
- Design responsive: Améliorer le design pour qu'il soit entièrement adaptatif, permettant une utilisation facile sur mobile, tablette et desktop.
- Notification météo: Mettre en œuvre une fonctionnalité envoyant des notifications aux utilisateurs lors de changements météorologiques importants (par exemple, une tempête ou des pluies approchantes).
- Optimisation des performances: Améliorer la réactivité de l'application en optimisant les appels à l'API ou en utilisant des techniques de mise en cache.

En résumé, cette application météorologique constitue une base fonctionnelle pouvant être enrichie avec diverses fonctionnalités pour offrir une expérience utilisateur plus poussée et plus interactive. Elle fournit une solide fondation pour explorer davantage les appels API en JavaScript et la création d'applications interactives.

Chapitre 22

Stockage de session en JavaScript

En termes simples, une session en informatique est la période au cours de laquelle un utilisateur se connecte à un site web. Par exemple, une “session” commence dès qu’un utilisateur ouvre un navigateur et entre dans un site en ligne et se termine une fois le navigateur fermé désinstallé. Le stockage de session en JavaScript diffère car il permet aux développeurs de stocker des informations spécifiques à un utilisateur pour la durée de la session. Cela signifie qu’une fois que l’utilisateur a accédé au site, les données sont accessibles pour ce cas utilisateur spécifique jusqu’à ce que la session soit terminée.

En somme, le **sessionStorage** en JavaScript est un objet qui aide à stocker temporairement les données directement sur le navigateur de l’utilisateur contrairement à localStorage qui stocke les données même après la fermeture du navigateur. Et très souvent, cette réserve est utilisée pour sauvegarder des informations temporaires comme les préférences d’affichage, les informations de connexion et les états temporaires de l’interface utilisateur. Cependant, il est à noter qu’allant de 5 à 20 Mo, la capacité de stockage dépend du navigateur.

Utilisation des méthodes sessionStorage

Les principales méthodes pour utiliser de stockage de session en JavaScript sont les suivantes:

sessionStorage.setItem(key, value) : stocke une valeur associée à une clé donnée dans la session.
sessionStorage.getItem(key) : récupère la valeur stockée associée à une clé donnée.
sessionStorage.removeItem(key) : supprime la clé et la valeur associée.
sessionStorage.clear() : efface les clés et valeurs stockées.

Exemple : **Stocker des données dans la session**
Pour stocker des données dans une session, on utilise la méthode setItem(). Par exemple :

```
sessionStorage.setItem("userName", "Jean");
```

Ce code stocke la chaîne de caractères "Jean" dans la session sous la clé "userName".

Récupérer les données d'une session
Pour récupérer des données stockées dans une session, on utilise la méthode getItem() :

```
let userName = sessionStorage.getItem("userName");
console.log(userName); // Affichera "Jean"
```

Cette ligne récupère la valeur associée à la clé "userName" et l'affiche dans la console.

Vérifier l'existence d'une session
Voici un exemple de code pour vérifier si une session existe :

```
    if (sessionStorage.length === 0) {
        console.log("La session n'existe pas.");
    } else {
        console.log("La session existe.");
    }
```

Le nombre de clés stockées dans **sessionStorage** est vérifié. Si aucune donnée n'est stockée (longueur égale à 0), cela signifie que la session n'existe pas.

Suppression des données de la session
Vous pouvez supprimer une donnée spécifique avec la méthode removeItem() :

```
sessionStorage.removeItem("userName");
let userName = sessionStorage.getItem("userName");
console.log(userName); // Affichera "null"
```

Cela supprime la clé `"userName"` ainsi que la valeur associée.

Exercice de pratique : Gestion des données avec un formulaire
Dans cet exercice, nous allons créer un formulaire simple demandant à l'utilisateur de saisir son prénom ainsi que son nom de famille, puis stocker ces informations dans une session. Une fois soumis, le formulaire redirigera l'utilisateur vers une autre page Web où les détails fournis seront affichés. Alternativement, nous pourrions demander d'autres types de renseignements personnels et professionnels à stocker de manière temporaire, comme la date de naissance ou l'emploi. Quelle que soit la méthode employée, l'objectif principal reste d'apprendre les bases de la gestion des sessions en ligne.
Formulaire HTML
Voici le formulaire HTML de la page principale (index.html) :

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta  name="viewport"  content="width=device-width,
initial-scale=1.0">
    <title>Exercice sur les sessions</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <form id="userForm">
        <div id="result"></div>
        <label for="name">Nom :</label>
        <input type="text" id="name" name="name"><br><br>
```

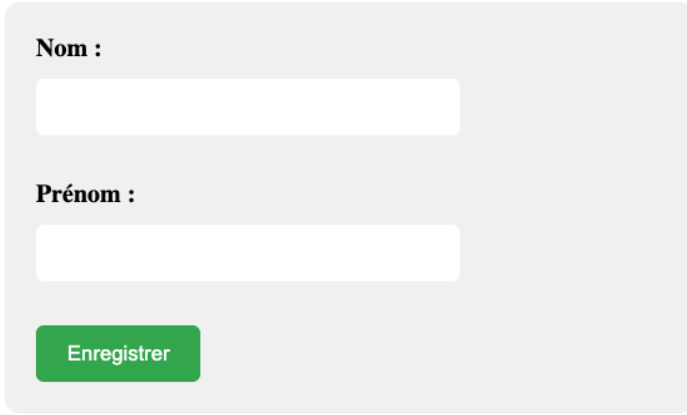
```
        <label for="surname">Prénom :</label>
                                <input    type="text"    id="surname"
name="surname"><br><br>

            <button  type="submit"  id="submit-btn">Enregistrer</
button>
    </form>
```

```
    <script src="my_script.js"></script>
</body>

</html>
```

Style CSS

Voici un exemple de style CSS pour mettre en forme le formulaire (style.css):

Résultat :



Script JavaScript

Dans le fichier my_script.js, nous allons gérer le stockage des informations et la redirection vers une autre page :

```
    // On récupère l'élément formulaire
```

```
    let form = document.getElementById("userForm");

    // Ajout d'un événement de soumission du formulaire
    form.addEventListener("submit", function (event) {
          event.preventDefault(); // Empêche la soumission par
défaut

        // Récupération des valeurs du formulaire
        let name = document.getElementById("name").value;
                                          let     surname     =
document.getElementById("surname").value;

        // Stockage des valeurs dans la session
        sessionStorage.setItem("userName", name);
        sessionStorage.setItem("userSurname", surname);

        // Redirection vers la page page2.html
        window.location.href = "page2.html";
    });
```

Page2.html

La page page2.html va récupérer les données stockées dans la session et les afficher :

```
    <body>
        <div id="message"></div>

        <script src="script2.js"></script>
    </body>
```

Script pour page2.html

Le fichier script2.js récupère les données de session et les affiche dans un message :

```
    // Vérifier si la session existe et afficher les données
    if (sessionStorage.length !== 0) {
        let name = sessionStorage.getItem("userName");
        let surname = sessionStorage.getItem("userSurname");
```

```
        let message = "Bonjour " + surname + " " + name + "
!";
            document.getElementById("message").textContent =
message;
    }
```

Conclusion

En somme, nous avons exploré le sessionStorage pour sauvegarder et récupérer des données temporaires avec JavaScript. Nous avons vu comment créer un formulaire HTML enregistrant des informations dans une session, rediriger l'utilisateur vers une autre page (page2.html) et afficher les données stockées. Ce mécanisme convient pour des sessions éphémères mais pourrait inclure des fonctionnalités plus poussées telles que la gestion de sessions via des jetons d'authentification ou la synchronisation entre onglets.

De nombreuses améliorations restent possibles comme l'ajout de champs supplémentaires au formulaire, l'implémentation d'une interface plus ergonomique ou l'intégration avec une base de données. L'objectif premier était de présenter simplement le fonctionnement du sessionStorage. À l'avenir, de telles fonctionnalités pourraient permettre le développement d'applications plus complexes manipulant des données temporaires côté client.

Conclusion générale

En arrivant à la fin de ce voyage à travers les multiples facettes de JavaScript, nous avons exploré ensemble l'évolution d'un simple script en une véritable application interactive. Ce livre vous a guidé depuis les bases, en passant par la gestion de variables, l'utilisation des opérateurs, jusqu'aux structures plus avancées comme les classes, l'héritage et la programmation asynchrone.

JavaScript, comme vous l'avez découvert, est bien plus qu'un langage de programmation. Il est l'outil qui, entre vos mains, peut transformer une idée en une application fonctionnelle, dynamique et connectée au monde extérieur. À travers des chapitres sur la manipulation du DOM, la communication avec des API, la gestion des événements et même des projets pratiques, vous avez acquis une compréhension à la fois solide et nuancée de ses possibilités.

Ce livre ne marque pas la fin de votre apprentissage, mais plutôt un tremplin vers des projets plus ambitieux. Vous avez désormais entre les mains les bases et les techniques nécessaires pour continuer à progresser et créer des applications toujours plus riches et performantes. Qu'il s'agisse de développer des applications pour le web, d'interagir avec des services en ligne ou de gérer des données utilisateur en temps réel, JavaScript vous offre des possibilités infinies.

N'oubliez pas que la programmation est un domaine en constante évolution. De nouvelles fonctionnalités, des bibliothèques et des frameworks continuent de voir le jour. Je vous encourage donc à continuer d'explorer, à vous lancer dans des projets personnels, et à maintenir votre curiosité en éveil.

Merci d'avoir pris ce chemin à mes côtés. En espérant que ce livre vous ait apporté autant de satisfaction dans l'apprentissage que j'en ai eu à le concevoir. À vous désormais de transformer vos idées en code et de laisser JavaScript révéler toute sa puissance à travers vos créations.

À propos de l’auteur

Alex Coder est un enseignant universitaire, développeur (PHP, JavaScript, Python, Java, C++ etc.) et Data scientist expérimenté dans le domaine des technologies numériques. Fort d'une expertise reconnue, il a dirigé de multiples formations et séminaires internationaux au sein d'institutions de renommée mondiale. Son approche pédagogique exceptionnelle lui a valu une réputation d'excellence parmi ses étudiants, grâce à sa capacité à simplifier des concepts complexes et à les rendre accessibles à tous.

Enseignant universitaire depuis plus de 20 ans, Alex Coder partage sa passion pour l'apprentissage et la transmission des connaissances en informatique, Data science et le Machine Learning. Sa longue expérience dans le domaine éducatif lui permet d'inspirer plusieurs générations d'apprenants à travers le monde, en leur transmettant non seulement des compétences techniques, mais aussi une vision globale du secteur du développement et d’analyse de données.

Passionné par le partage des connaissances, Alex Coder excelle dans la transmission de compétences en informatique, inspirant ainsi plusieurs générations d'apprenants à travers le monde. En plus de ses activités pédagogiques, il offre des formations spécialisées aux professionnels, les aidant à perfectionner leurs compétences et à rester à la pointe de leur domaine respectif.

Son dévouement à l'excellence, sa passion pour l'enseignement et son expertise technique font de Alex Coder une référence incontournable dans le domaine du développement web et de la formation numérique.

Lien de téléchargement

Lien de téléchargement du code source des solutions proposées pour tous les exercices et les projets réalisés dans ce livre :

https://www.skilldeve.com/codesourcejavascript

Notes

[←1] https://code.visualstudio.com/

[←2]

https://www.jetbrains.com/webstorm/

[←3]

https://visualstudio.microsoft.com/fr/vs/community/

[←4] https://atom.io/

[←5]
https://www.sublimetext.com/

[←6] https://notepad-plus-plus.org/

[←7]

https://www.vim.org/

[←8]

https://www.gnu.org/software/emacs/

[←9]

https://fr.javascript.info/article/devtools/bug.html